NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLICATION OF OPTICAL FLOW SENSORS FOR DEAD RECKONING, HEADING REFERENCE, OBSTACLE DETECTION, AND OBSTACLE AVOIDANCE

by

Tarek M. Nejah

September 2015

Thesis Advisor: Zachary Staples
Co-Advisor: Clark Robertson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Electrical Engineer's thesis	
4. TITLE AND SUBTITLE APPLICATION OF OPTICAL FLOW SENSORS FOR DEAD RECKONING, HEADING REFERENCE, OBSTACLE DETECTION, AND OBSTACLE AVOIDANCE			**5. FUNDING NUMBERS**
6. AUTHOR(S) Nejah, Tarek M.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

A novel approach for dead reckoning, heading reference, obstacle detection, and obstacle avoidance using only one optical mouse sensor was presented in this thesis. Odometry, position tracking, and obstacle avoidance are important issues in mobile robotics. Traditional odometry and motion-tracking sensors provide relative displacement data on a frame-to-frame basis, and they are usually mounted in arrays to provide accurate measurements with small estimation errors. Optical flow sensors stand as a tempting solution for robot self-localization and dead reckoning. In this work, using only one inexpensive optical mouse sensor, we were able to perform optical odometry, dead reckoning, and heading reference. Also, obstacle detection and avoidance remains a challenging area of research. Most of the existing works are based on stereo imaging and computation of the time-to-contact. These techniques are complex and usually require the use of more than one vantage point. The use of one optical mouse sensor as an obstacle-detection sensor was proposed in this work. The detection process is simple and is based on the surface-quality factor variation. As far as we know, no one has ever used this technique to perform obstacle-detection and avoidance. Using one sensor for motion tracking and one sensor for object detection in association with an Arduino microcontroller, we built an indoor ground robot capable of environment sensing, obstacle avoidance, and position tracking. The behavior of the robot can be monitored from a remote station. The experimental results obtained were promising and can be further improved.

14. SUBJECT TERMS robotics, optical flow sensors, optical flow techniques, optical mouse sensors, dead reckoning, position tracking, obstacle detection, obstacle avoidance.			**15. NUMBER OF PAGES** 153
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

APPLICATION OF OPTICAL FLOW SENSORS FOR DEAD RECKONING, HEADING REFERENCE, OBSTACLE DETECTION, AND OBSTACLE AVOIDANCE

Tarek M. Nejah
Captain, Tunisia Air Force
National Diploma of Engineer in Telemechanic, Aviation School of Borj El Amri, 2005

Submitted in partial fulfillment of the
requirements for the degree of

ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Approved by: Zachary Staples
Thesis Advisor

Clark Robertson
Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A novel approach for dead reckoning, heading reference, obstacle detection, and obstacle avoidance using only one optical mouse sensor was presented in this thesis. Odometry, position tracking, and obstacle avoidance are important issues in mobile robotics. Traditional odometry and motion-tracking sensors provide relative displacement data on a frame-to-frame basis, and they are usually mounted in arrays to provide accurate measurements with small estimation errors. Optical flow sensors stand as a tempting solution for robot self-localization and dead reckoning. In this work, using only one inexpensive optical mouse sensor, we were able to perform optical odometry, dead reckoning, and heading reference. Also, obstacle detection and avoidance remains a challenging area of research. Most of the existing works are based on stereo imaging and computation of the time-to-contact. These techniques are complex and usually require the use of more than one vantage point. The use of one optical mouse sensor as an obstacle-detection sensor was proposed in this work. The detection process is simple and is based on the surface-quality factor variation. As far as we know, no one has ever used this technique to perform obstacle-detection and avoidance. Using one sensor for motion tracking and one sensor for object detection in association with an Arduino microcontroller, we built an indoor ground robot capable of environment sensing, obstacle avoidance, and position tracking. The behavior of the robot can be monitored from a remote station. The experimental results obtained were promising and can be further improved.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

UAV	Unmanned Aerial Vehicle
GPS	Global Positioning System
INS	Inertial Navigation System
2D	Two Dimensions
3D	Three Dimensions
SPI	Serial Peripheral Interface
IAS	Image Acquisition System
DSP	Digital Signal Processor
NCS	Non Chip Select
NPD	Non Power Down
SCLK	Serial Clock
MOSI	Master-Out-Slave-In
MISO	Master-In-Slave-Out
MSB	Most Significant Bit
LSB	Least Significant Bit
SQUAL	Surface Quality
PWM	Pulse Width Modulation
IDE	Integrated Development Environment
DAC	Digital to Analog Converter
SS	Slave Select
WSN	Wireless Sensor Network
ISM	Industrial Scientific and Medical
UART	Universal Asynchronous Receiver Transmitter
FMS	Flight Management System
FMC	Flight Management Computer
CDU	Control/Command Display Unit
SONAR	Sound Navigation and Ranging
LIDAR	Light Detection and Ranging

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Inspired by the way flying insects rely on optical estimation for landing, obstacle avoidance, distance estimation, and speed regulation, optical flux has been, for decades, an interesting area of research for many robotics researchers. Optical flow techniques present an effective solution to navigation and environmental interaction problems that many ground and aerial robots encounter. To be more specific, the integration of optical flux in robotics allows accurate measurements of traveled distance, altitude, and velocity. Different optical flow algorithms (Lucas-Kanade method, Horn-Schunck method, Image interpolation method, Block matching algorithm, etc.) have been generated and adopted to mimic the behavior of flying insects. A variety of optical flow sensors have been used. The most popular sensors are optical mice, omnidirectional vision systems, and binocular vision systems. Motion field-estimation models such as Pin-Hole Image Plane and Spherical Imaging Surface provide rotational velocities, translational velocities, and terrain information expressed in the camera body frame; however, this work shows that optical flow can also be estimated from an inexpensive optical mouse sensor with a narrow field of view.

A. THESIS OBJECTIVES

Usually, in a typical ground based robot, shaft encoders mounted to the wheels provide motion estimation. These encoders are set to measure the speed of the platform and the distance run. Unfortunately, wheels can easily slip on uneven surfaces or when colliding with obstacles. This phenomenon results in position and distance estimation errors. Odometry and dead reckoning rely on the data generated by motion sensors to estimate change in position over time. As a small and inexpensive optical flow sensor, the optical mouse turns out to be a good solution for these problems. An optical mouse or a camera navigation system has no moving parts, no contact with the floor, and does not care about the sliding effect of the wheels. As a result, position estimation and distance measurements can be accurate. In this thesis, an optical flow sensor is implemented as an optical odometer and a dead reckoning sensor in a ground mobile indoor robot. We

determine the traveled distance and the instantaneous position of the vehicle from the generated data delivered by the optical sensor to an Arduino microcontroller. The microcontroller controls the speed, heading of the robot, and flow of data coming in and out from the different sensors used. In addition to odometry and dead reckoning, obstacle detection and avoidance or environment recognition using optical flow sensors is one of the objectives of this thesis. In addition, robotics and control system implementation in embedded systems are explored.

B. ORGANIZATION

The second chapter involves a discussion of some optical flow algorithms including Lucas-Kanade, Horn-Schunck and others that have been generated and adopted to mimic the behavior of flying insects. We briefly discuss the different motion field-estimation models such as Pin-Hole Image Plane and Spherical Imaging Surface. The different computations and calculations related to most optical flow sensors such as optical mice, omnidirectional vision systems and binocular vision systems are addressed. Also, we briefly discuss the different optical flow sensors commonly used in robotics for dead reckoning and distance measurement.

In the third chapter, we introduce optical flow sensors as a solution to the problems met by typical shaft encoders and optical encoders. We explain how optical flow techniques can improve dead-reckoning performances of ground controlled robots. We deal with the different rotation matrices, translations, and homogeneous transformations used to express the position and velocity of the robot relative to the different frames involved (i.e., world frame, robot frame, and sensor frame).

The experimental part of the thesis is described in Chapter IV, where all the steps and methodologies considered to successfully implement and use an optical mouse sensor as an optical odometer, dead-reckoning, and obstacle-detection and avoidance sensor in a ground mobile indoor robot are described. All the components used in the experimental part are described in detail (Arduino Due, ADNS 3080, XBEE PRO 90 transceiver, DC motors board, etc.).

Finally, the work accomplished and results obtained are summarized in Chapter V, and some ideas of how this work can be further improved are provided.

C. BENEFIT OF STUDY

A systematic investigation into optical flow sensor-based-robotics navigation systems is presented. This work may be considered in the future as a framework for further studies and investigations concerned with solving the navigation and obstacle avoidance problems encountered by ground and aerial robots.

THIS PAGE INTENTIONALLY LEFT BLANK

II. OPTICAL FLOW OVERVIEW

The majority of unmanned aerial vehicles (UAVs) are equipped with Global Positioning Systems (GPSs) and inertial navigation systems (INSs). With no ability to sense terrain features, these navigation systems cannot offer a safe and optimized mode of operation when navigating in GPS degraded zones or indoor environments [1]. One of the main problems faced in the development of fully automated vehicles (ground and aerial robots) is the perception of the environment; thus, it is necessary to be able to detect still and moving objects in order to reduce risk of collision. Motion estimation is a process that involves studying moving objects in a video sequence, seeking the correlation between two back-to-back images to predict the change in position of the content. There are several motion estimation methods. The most widely used methods are optical flow techniques. Optical flow is a visual displacement field that explains variations in a moving image in terms of image points.

A. OPTICAL FLOW DEFINITION

Optical flow can be expressed as a function of image pixels (Apparent Motion Definition) or a function of azimuth and elevation angles (Motion Field Definition), as stated in [2].

1. Apparent Motion Definition

In robotics, optical flow can be perceived as an object's apparent motion from the eye of an embedded camera and can be computed as the difference between two successive images and expressed according to [3] as

$$[\dot{x}, \dot{y}]^T = f(x, y). \tag{0}$$

The optical flow here is represented as the relative displacements in the x and y directions over a time t, and (x,y) represent any point on the image plane. The unit for the point motion can be pixels per frame or pixels per seconds.

2. Motion Field Definition

Optical flow can also be perceived as the relative three-dimensional (3D) motion between the camera and the scene into the image plane. The relative motion can be represented by different motion field models. A simplified optical-flow motion field model is described in Figure 1. The optical-flow motion field can be expressed, as shown by Haiyang et al. [3], as

$$OF_m = \frac{V}{d}. \tag{2}$$

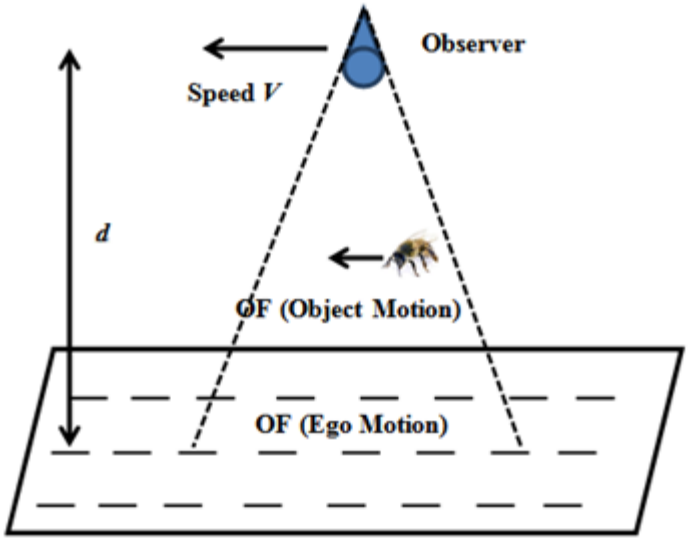


Figure 1. Definition of optical flow (after [3]).

The observer speed is denoted by ***V***. The distance separating the observer from the object along the optical axis is denoted by ***d***. The optical flow is expressed in radians per seconds or degrees per seconds. The movement of the observer relative to the static environment is the Ego Motion (EM); whereas, the Object Motion (OM) stands for the displacement of independent objects. As a result, the optical-flow field contains information for both EM and OM.

B. OPTICAL FLOW COMPUTATION ALGORITHMS

Thanks to the huge progress made in image processing and computer vision, many algorithms have been adopted to determine optical flow from two consequent images. According to [4] and [5], we find that most of these algorithms operate under specific assumptions that can be mathematically written as

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{3}$$

and

$$I_x \, \dot{x} + I_y \, \dot{y} + I_t = 0 \,, \tag{4}$$

where the light intensity or image brightness of a point (x,y) on the two-dimensional (2D) image plane at time t is here denoted by $I(x,y,t)$, and I_x, I_y, and I_t are, respectively, the partial derivatives of the intensity function with respect to x, y, and t. Equation (3) implies that local variations of the image intensity can only be caused by the movement of the object with respect to the observer. Equation (4) implies that the motion over a tiny neighborhood of pixels is uniform. In robotics, two of the most popular optical-flow-intensity algorithms are the Lucas-Kanade and the Horn-Schunk algorithms. Other methods based on features other than intensity can be used to compute optical flow but are not addressed here because they are not as popular as the Lucas-Kanade and Horn-Schunk algorithms. A survey of the different optical flow algorithms can be found in [2] and [6].

1. Lucas-Kanade Method

The Lucas-Kanade method is a differential method used for OF estimation. This method was developed by Bruce D. Lucas and Takeo Kanade. It presumes the flow is constant around a considered pixel **p** and solves the equation of optical flow for all pixels in that neighborhood using the least squares method [7]. The optical flow equations may be applied for all the pixels belonging to a window of center p. Considering a window of n pixels (q_1, q_2,…,q_n), we see that the local velocity or image flow vector $V = (\dot{x}, \dot{y})^T$ must satisfy

$$
\begin{aligned}
&I_x(q_1)\,\dot{x} + I_y(q_1)\;\dot{y} = -I_t(q_1)\\
&I_x(q_2)\,\dot{x} + I_y(q_2)\;\dot{y} = -I_t(q_2)\\
&\vdots\\
&I_x(q_n)\,\dot{x} + I_y(q_n)\;\dot{y} = -I_t(q_n)
\end{aligned}. \tag{5}
$$

In matrix representation, these equations can be written as

$$
\begin{matrix} A & . & V & = & B \end{matrix}
$$

$$
\begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix}
\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}
=
\begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \vdots \\ -I_t(q_n) \end{bmatrix}. \tag{6}
$$

By rearranging the matrix form shown above, we express the image flow vector ***v*** as

$$
V = (A^T A)^{-1} A^T B \tag{7}
$$

$$
\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}
=
\begin{bmatrix} \sum_{i=1}^{n} I_x(q_i)^2 & \sum_{i=1}^{n} I_x(q_i) I_y(q_i) \\ \sum_{i=1}^{n} I_y(q_i) I_x(q_i) & \sum_{i=1}^{n} I_y(q_i)^2 \end{bmatrix}^{-1}
\begin{bmatrix} -\sum_{i=1}^{n} I_x(q_i) I_t(q_i) \\ -\sum_{i=1}^{n} I_y(q_i) I_t(q_i) \end{bmatrix}. \tag{8}
$$

1. Horn-Schunk Method

The difference between the Lucas-Kanade and Horn-Schunk methods is that the first algorithm assumes smoothness in the flow over a small set of pixels; however, the second algorithm assumes that the motion is uniform over the whole image. As a result, the Horn–Schunck algorithm results in a higher density of flow vectors than the Lucas-Kanade algorithm; however, it is much more sensitive to noise [6]. In the Horn-Schunk method, the flow for a 2D image stream is presented as a global energy function that must be minimized [8], that is,

$$
E = \iint \left\langle \left(I_x u + I_y v + I_t \right)^2 + \alpha^2 \left(\nabla^2 u + \nabla^2 v \right) \right\rangle dx dy\,, \tag{9}
$$

where

$$u = \dot{x} = \frac{dx}{dt} = f(x,y) \quad ; \quad v = \dot{y} = \frac{dy}{dt} = f(x,y) \ , \tag{10}$$

and

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} . \tag{11}$$

The optical flow vector is represented by $V = (u,v)^T$. Larger values for ***α***, the constant of regularization, result in smoother flows. Using the Finite Differences Method, we can approximate the Laplacians of ***u*** and ***v***, respectively,

$$\nabla^2 u = \bar{u}(x,y) - u(x,y) \tag{12}$$

and

$$\nabla^2 v = \bar{v}(x,y) - v(x,y) . \tag{13}$$

The quantities $\bar{u}(x,y)$ and $\bar{v}(x,y)$ are, respectively, two weighted averages of *u* and *v* computed in the surrounding area of the (*x*,*y*) point. By solving the associated Multi-Dimensional Euler-Lagrange equations, we can minimize the general expression of the flow, with the result [8]

$$I_x \left(I_x u + I_y v + I_t \right) - \alpha^2 \nabla^2 u = 0 \tag{14}$$

and

$$I_y \left(I_x u + I_y v + I_t \right) - \alpha^2 \nabla^2 v = 0 . \tag{15}$$

By substituting (12) and (13), respectively, into (14) and (15), we get

$$\left[{I_x}^2 + \alpha^2 \right] u + I_x I_y v = \alpha^2 \bar{u} - I_x I_t \tag{16}$$

and

$$I_x I_y u + \left[{I_y}^2 + \alpha^2 \right] v = \alpha^2 \bar{v} - I_y I_t . \tag{17}$$

The pair of linear equations above is true for each point in the image; however, since the neighboring values of the flow field are involved, an iterative solution must be considered:

$$u^{k+1} = \bar{u}^k - \frac{I_x \left(I_x \bar{u}^k + I_y \bar{v}^k + I_t \right)}{{I_x}^2 + {I_y}^2 + \alpha^2} \quad ; \quad v^{k+1} = \bar{v}^k - \frac{I_y \left(I_x \bar{u}^k + I_y \bar{v}^k + I_t \right)}{{I_x}^2 + {I_y}^2 + \alpha^2} \tag{18}$$

As stated earlier, many algorithms have been adopted to compute optical flows; however, this is not possible without the availability of some optical flow motion field estimation models capable of projecting 3D relative motions on a 2D image plane.

C. OPTICAL FLOW MOTION FIELD ESTIMATION MODELS

Whether the objects are moving in the scene or the observer is moving through the scene, optical flow allows movement detection. Essentially, there are two main approaches for the derivation of the optical flow motion field estimation models. The first one is the pin-hole image plane approach, and the second one is the spherical imaging surface approach. The optical flow motion-field estimation models take care of the projection of a relative 3D motion onto a 2D image plane. Optical flow sensing is mainly realized by considering a camera as a sensor. Let us suppose that a moving camera takes two successive images, one at time t and the other at time t+1. The two images are compared to each other to translate all the information relative to rotational velocities, translational velocities, and surface information. From Figure 2 and Figure 3, it can be seen that the resulting optical flow field is not the same when the camera is moving. In this case, the optical flow field contains information about the observer translational velocity. This is usually how dead reckoning and optical odometry for mobile robots equipped with optical flow sensors works.

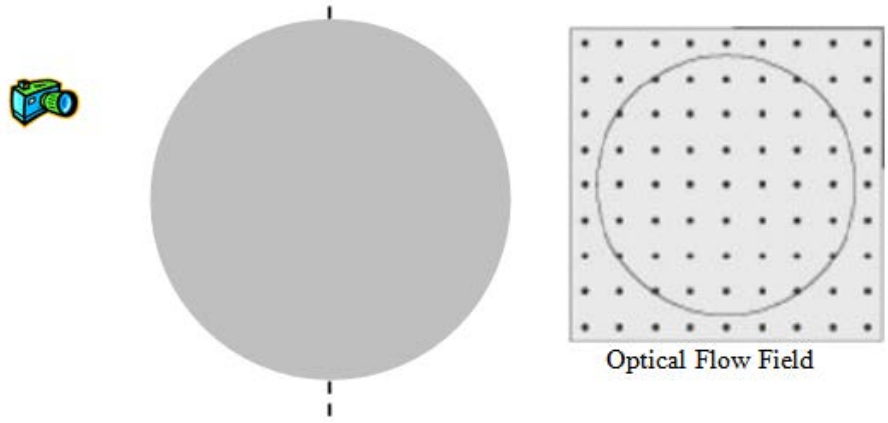


Figure 2. Optical flow field estimated by a non-moving observer.

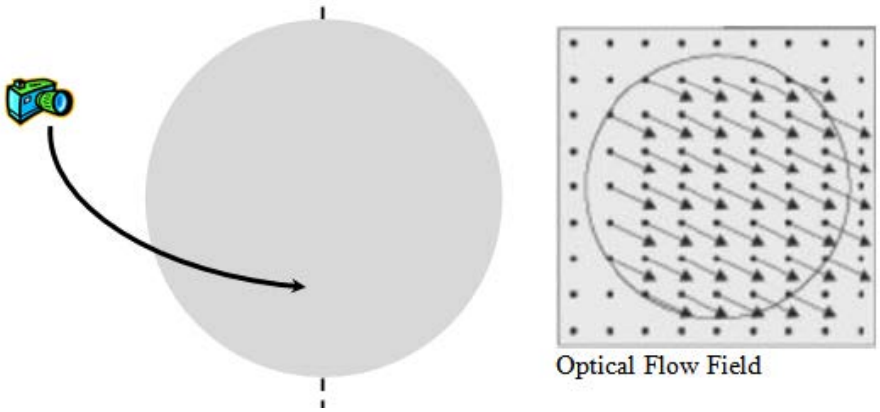


Figure 3. Optical flow field estimated by a moving observer.

Pin-hole image plane and spherical imaging surface approaches are described in detail in [3].

D. APPLICATION OF OPTICAL FLOW SENSORS

In robotics, optical flow sensors have been widely used as navigation sensors mounted to indoor and outdoor robots to complete a variety of navigation tasks. Capable of keeping track of any displacement, optical flow sensors have been used as optical odometers to ensure accurate measurement of distance. Even though typical encoders such as shaft and optical encoders present a considerable margin of error when dealing with distance measurement, optical flow sensors have not been able to totally replace them due to redundancy issues.

Another application of optical flow sensors is obstacle avoidance. Researchers are motivated to use optical flow sensors as obstacle avoidance sensors due to the fact that they have a wide field-of-view. By only mounting a few of them on a robot, we can cover all of the surrounding area.

Altitude hold is a new application of optical flow sensors. Some researchers used them as a direct feedback to micro UAVs to maintain a specific altitude and control the yaw angle. Under degraded GPS performance, optical flow sensors can also stand as solutions for dead reckoning. They can give accurate estimates of current position and

speed based on previous calculations. In addition, combined with inertial navigation systems, they can provide precise measurements of height, altitude, and horizontal and ground velocities which can be used for hovering control. Inspired by honeybees' grazing landing, some researchers have demonstrated the possibility of integrating optical flow sensors for stabilization and landing on fixed and moving platforms. These applications are very attractive when it comes to military deployments and emergency landings. A summary of the different applications of optical flow sensors and some of the works already done is provided in Table 1.

Table 1. Optical flow based navigation works and approaches (after [3]).

Navigation functions	Authors	Robotic platform	OF computation technique
Landing on moving platform	ONERA-UNICE-ANU	Quad-rotor	Lucas-Kanade algorithm
Velocity and height estimation	UNSW-ADFA-UTS	Helicopter	Image interpolation algorithm
Obstacle avoidance		Flying wing	Optical mouse sensor
Altitude keeping	EPFL	Ultra-light MAV	Image interpolation algorithm
Estimation hovering	UAEH-UTC	Eight-rotor VTOL	Lucas-Kanade algorithm
Velocity estimation	ETHZ	Quad-rotor	Block matching algorithm
OF comparison: vision vs. navigation sensors	WVU	Small fixed-wing	Sift feature

III. DEAD RECKONING AND ODOMETRY FOR INDOOR ROBOTS USING AN OPTICAL MOUSE SENSOR

Dead reckoning is the estimation of position and can also be referred to as self-localization or position tracking. Odometry is the estimation of speed and distance. In the case of ground robots, sensors are usually attached to the wheels, and the collected data is analyzed to estimate the motion of the robot. The most popular and widely used sensors are the absolute and incremental rotary optical encoders. Unfortunately, these sensors generate unbounded errors, especially when paths are not straight. Inertial sensors are also used for dead-reckoning and odometry applications, but they suffer from the same type of errors. One factor behind the lack of precision of rotary encoders is that when a wheel-based robot slips, the wheels do not spin, leading to zero data collected by the sensors. This is the major handicap for indoor and outdoor robots; therefore, sensors based solely on optical flow computation must be adopted in robotics to solve the problems of inaccuracy and unreliability of traditional sensors. Among all the optical flow sensors, the use of an array of high speed optical flow mice has been proposed as a solution for dead reckoning and odometry issues [9]–[10].

A. ROTARY DIGITAL OPTICAL ENCODERS

An optical encoder is an electronic device that converts the angular position or motion of an axle to a digital code or sequence of pulses. The optical encoder's disc is a glass or plastic disc containing transparent and dark spots. A light source emits the light. Depending on whether the light reflects over the white surface or the black surface of the disc, we see that the photo detectors detect the optical pattern resulting from the disc's position. Optical encoders usually consist of infrared emitting diodes and NPN phototransistors. The emitting diode and detector are mounted side-by-side on parallel axes. The code collected is then converted by a microcontroller to an absolute or relative position measurement. In the case of absolute encoders, a unique digital word corresponds to a specific rotation of the shaft; however, as the shaft rotates, an

incremental encoder generates digital pulses leading to the measurement of a relative position. An example of a rotary optical encoder is displayed in Figure 4.

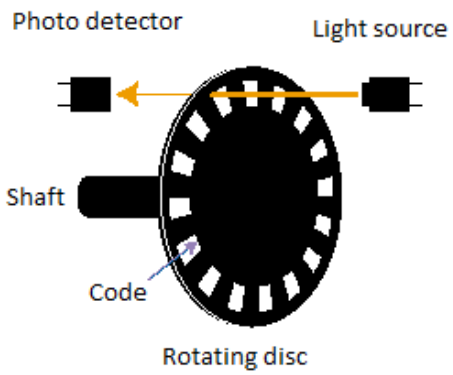


Figure 4. Example of rotary optical encoder (from [11]).

1. Absolute Encoder

There are two popular types of absolute encoder: the gray and binary code encoders. From Figure 5, it can be seen that the main difference resides in the arrangement of dark and white spots. In the following, we consider a three-bit-digital-word absolute encoder; thus, we need three emitting diodes and three photo detectors mounted in parallel axes. The conversion from binary word to angle rotation is shown in Table 2.

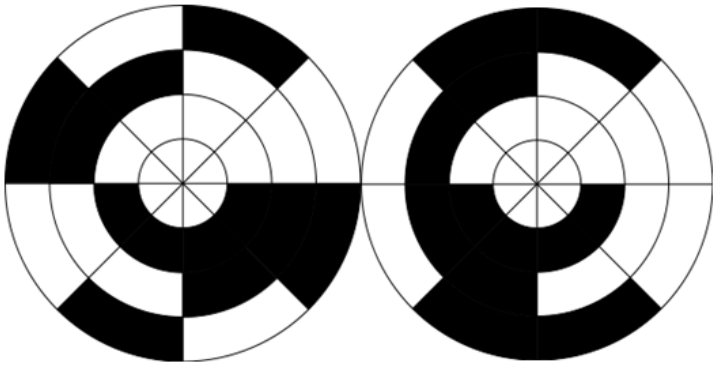


Figure 5. Binary and Gray encoding disc (from [12]).

Table 2. Three-bit digital word-to-angle conversion of an absolute encoder.

Decimal code	Rotation range in degrees	Binary code	Gray code
0	0 to 45	000	000
1	45 to 90	001	001
2	90 to 135	010	011
3	135 to 180	011	010
4	180 to 225	100	110
5	225 to 270	101	111
6	270 to 315	110	101
7	315 to 360	111	100

For a three bit digital word, we get 2^3=8 angle rotations or distinct shaft positions. For a n-bit digital word, we have 2^n distinct angle rotations; thus, increasing the number of bits per digital word, significantly increases the precision of the position measurement. The gray code is preferred over the binary code since the uncertainty during one transition is always one bit.

2. Incremental Encoder

As shown in Figure 6 and Figure 7, the incremental or relative encoder has two sensors whose outputs are considered channels and called, respectively, channel A and channel B. The two output channels, A and B, are in quadrature, meaning they are 90 degrees out of phase. Waveforms A and B are decoded to produce a count-up pulse or a countdown pulse. Often, an additional output channel (INDEX) is added to count full revolutions. It is also used to define the zero position.

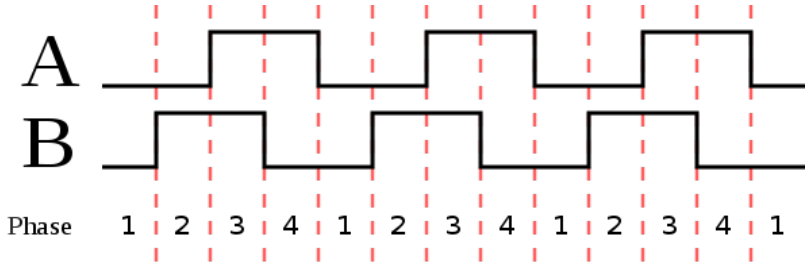


Figure 6. Example of channels A and B outputs (from [13]).

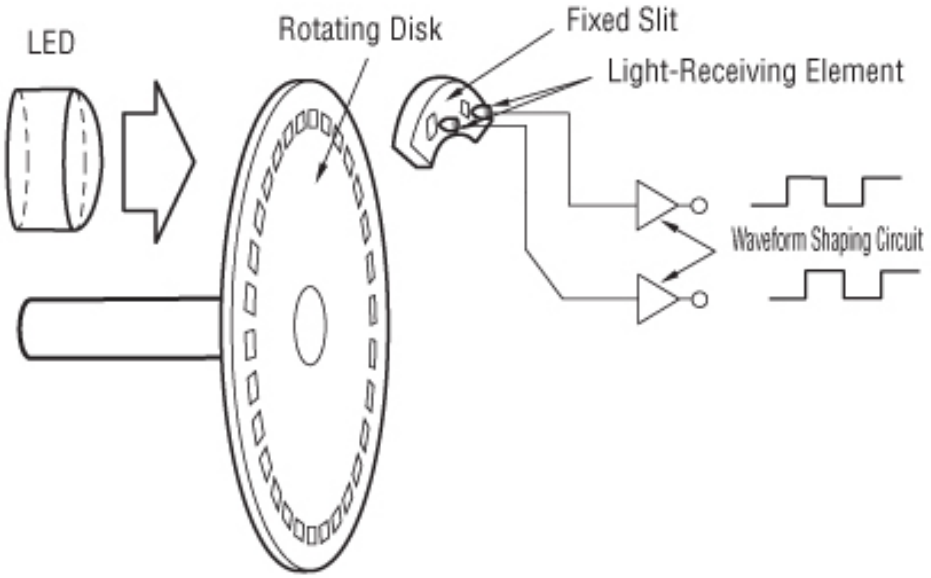


Figure 7. Incremental encoder (from [14]).

The principle of operation of an incremental encoder is illustrated in Table 3. For example, if the last value collected from A and B was 00 and the current value is 01, it means the wheel rotated a half step in the clockwise direction. Steps refer to the angle slots available on the wheel. By counting the number of steps the wheel rotated, we can determine precisely the position of the wheel at any time. The velocity can be determined from the angle of rotation and the time taken to perform the rotation. Generally, incremental encoders are preferred over absolute encoders since they give better results in term of precision and accuracy, with fewer electronic components involved.

Table 3. Example of output channels state diagram.

Phase	Clockwise rotation		Counter-clockwise rotation	
	A	**B**	**A**	**B**
1	0	0	1	0
2	0	1	1	1
3	1	1	0	1
4	1	0	0	0

B. OPTICAL MOUSE SENSORS

The idea of using optical mice as optical flow sensors is a powerful lure for robotics researchers due to several factors. First of all, mouse chips are capable of tracking 2D motions at very high resolutions. Second, they are small, light, and easy to mount on any robotics platform. Finally, the chips are abundant on the market and are inexpensive compared to the other optical flow sensors. The common problem of optical mouse sensors is that they are mainly designed to work on surfaces located a few millimeters from the sensor. Consequently, research was done to allow the application of these sensors where the platform or the robot is farther than a few millimeters from the tracking terrain. To accomplish this, optical imaging systems used in optical mice were modified with non-standard lenses, allowing the refocusing of light onto the sensor. The lenses used differ from one application to another according to the distance required between the sensor and the ground. Actually, this approach has been proven to work for different distances ranging from 2.0 cm above the surface for wheel-based robots to tens of meters for flying robots [15]–[16].

Three main factors need to be considered when selecting an optical mouse sensor for a certain application. These factors are the frame rate, the image size, and the resolution. The number of snapshots the sensor is capable of taking per second represents the frame rate. The higher the frame rate of the sensor, the greater is its ability to detect small motions. That means increased capabilities to track high speed movements. High

speed motion tracking also depends on the image size. That means if two sensors have the same frame rate, the one capable of taking larger images gives better results. For example, a 30 × 30-pixel-image sensor is better than an 18 × 18-pixel-image sensor. Finally, the last characteristic to consider in an optical mouse sensor is the resolution. The resolution of a sensor is usually expressed in counts per inch (cpi) and reflects the number of steps the sensor reports during a displacement of one inch. In other words, a high resolution sensor of 1600 cpi detects more surface details than a low resolution sensor of 400 cpi. The basic principles of operation for optical mice are described in detail in the next chapter as we introduce all the electronic components used to perform the experimental part of this thesis. The performances of different Avago-brand mouse chips are compared in Table 4.

Table 4. Characteristics of some Avago mouse-chip sensors.

Name	Type	Rated resolution (cpi)	Rated speed (inches/s)	Frame rate (fps)	Image size (pixels)
ADNS-2610	Optical	400	12	1500	18×18
ADNS-2051	Optical	400/800	14 at 1500 fps	500-2300	18×18
ADNS-5060	Optical	1050	30	-	19×19
ADNS-3080	Optical	400/1600	40 at 6400 fps	500-6469	30×30
ADNS-7050	Laser	800	20	-	22×22
ADNS-9500	Laser	5000	150	-	30×30

C. MOTION TRACKING

In robotics, three main reference frames are used to translate the motion of an object within a specific space. The first is the *world* frame, referred to as **W** frame. The second is the *robot* frame, referred to as **R** frame. The final one is the *sensor* frame, referred to as **S** frame. It is important to consider that the number of **S** frames depends on the number of sensors used since every sensor has its own frame. Let us consider the configuration of frames shown in Figure 8.

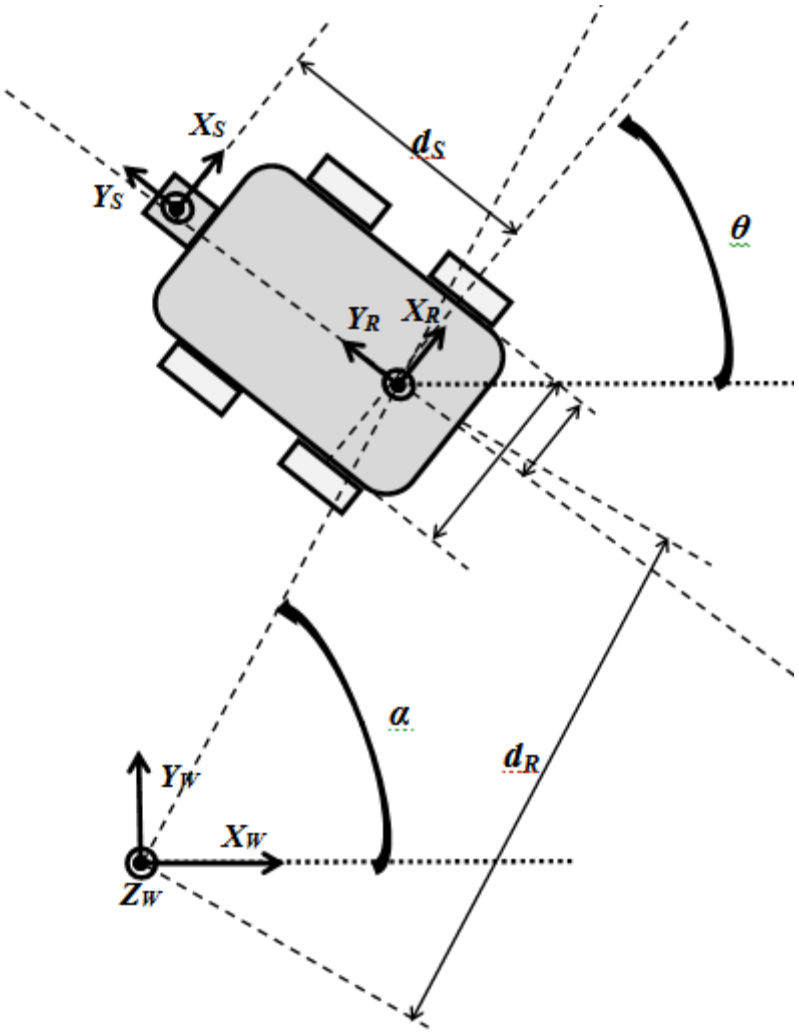


Figure 8. Example of reference frames configuration.

At the beginning, or at ***t*=0**, the **W** and **R** frames coincide. This means that the sensor and robot positions relative to the **W** frame are, respectively,

$$ {}^{W}O_S = {}^{R}O_S = \begin{pmatrix} 0 \\ d_S \\ 0 \end{pmatrix} \quad ; \quad {}^{W}O_R = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{19} $$

As the robot moves, the position of the robot frame and sensor frame relative to the absolute frame change. Using rotation matrices and homogeneous transformations, we

can represent all possible motion of the robot with respect to the **W** frame. The rotation matrix of frame **R** relative to frame **W** is given by

$$ {}^{W}_{R}R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{20} $$

Also, the rotation matrix of frame **S** relative to frame **R** is given by

$$ {}^{R}_{S}R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = I\,. \tag{21} $$

The robot position represented in the **W** frame system can be written as

$$ {}^{W}O_{R} = \begin{pmatrix} d_R \cos(\alpha) \\ d_R \sin(\alpha) \\ 0 \end{pmatrix}, \tag{22} $$

and the sensor position represented in the **R** frame system can be written as

$$ {}^{R}O_{S} = \begin{pmatrix} 0 \\ d_S \\ 0 \end{pmatrix}. \tag{23} $$

Using the different rotation matrix and positions of frames relative to each other, we get the corresponding homogeneous transformations

$$ {}^{W}_{R}T = \begin{pmatrix} & {}^{W}_{R}R & & {}^{W}O_{R} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & d_R \cos(\alpha) \\ \sin(\theta) & \cos(\theta) & 0 & d_R \sin(\alpha) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{24} $$

and

$$ {}^{R}_{S}T = \begin{pmatrix} & {}^{R}_{S}R & & {}^{R}O_{S} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d_S \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{25} $$

The position of the sensor relative to the world frame at any time ***t*** is given by

$$ {}^{W}P = {}^{W}_{R}T \times {}^{R}P = {}^{W}_{R}T \times \begin{pmatrix} {}^{R}O_S \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{W}O_S \\ 1 \end{pmatrix}, \tag{26} $$

which can be written

$$ {}^{W}P = {}^{W}_{R}T \times {}^{R}P = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & d_R\cos(\alpha) \\ \sin(\theta) & \cos(\theta) & 0 & d_R\sin(\alpha) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ d_S \\ 0 \\ 1 \end{pmatrix}. \tag{27} $$

Now, the sensor's position coordinates relative to the world frame are

$$ {}^{W}O_S = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -d_S\sin(\theta) + d_R\cos(\alpha) \\ d_S\cos(\theta) + d_R\sin(\alpha) \\ 0 \end{pmatrix}. \tag{28} $$

For each single motion of the robot (translation or rotation) we must solve these equations to know the position of the robot. With only two equations and three unknowns, it is difficult to determine the position of the robot. That is why, usually, multiple sensors are mounted on a moving platform. With only one sensor, dead reckoning is a complex problem to solve in robotics. In the next chapter, we present our solution that not only makes dead reckoning and odometry with only one sensor possible, but also remove the complexity of jumping from one frame to another by directly representing the position of the robot in the **W** frame.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTAL SETUP AND RESULTS

The main goal of this work is the design and development of an efficient, inexpensive, reliable, and simple obstacle avoidance and dead reckoning optical flow sensor that can be used in varying light conditions. All the steps taken to make this work achievable are outlined in this chapter. The work done throughout this thesis involved commercially available components, C++ coding, and custom algorithms. The components used to realize the experimental part of this thesis are the ADNS-3080, the Arduino Due, a DC motor board connected to four motors, a Parallax servo motor, and two XBEE modules.

A. HIGH PERFORMANCE OPTICAL MOUSE SENSOR ADNS-3080

The ADNS-3080 belongs to the family of ADNS optical mouse sensors manufactured by Avago Technologies. It is considered to be a high performance optical flow mouse sensor due to its key features that include the following [17]:

- Up to 40-inches per second (ips) and 15-g speed motion detection
- 500 to 6469-frames per second (fps) programmable frame rate
- 400 or 1600-counts per inch (cpi) selectable resolution
- 30 × 30 pixels image size

Accessing the sensor for data communication is possible through a four-wire Serial Peripheral Interface (SPI). The ADNS-3080 consists of an Image Acquisition System (IAS), a Digital Signal Processor (DSP), and a serial port. The IAS includes a tiny camera, a lens, and an illumination system. The DSP processes the microscopic terrain images captured by the IAS and determines the distance, the direction, and the Δx and Δy relative displacements. An external microcontroller can be used to access (read/write) the different registers of the sensor by using the SPI. The block diagram and pinout of the ADNS-3080 are shown in Figures 9 and 10, respectively.

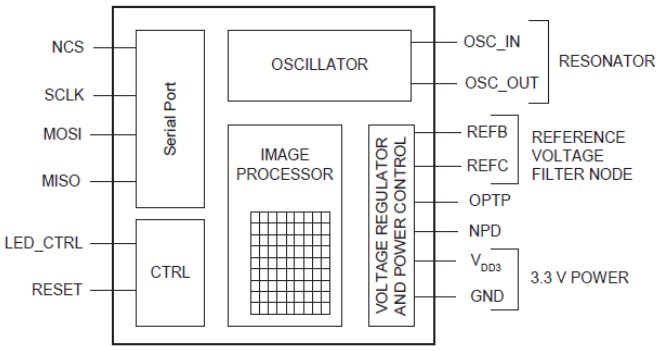


Figure 9. Block Diagram of ADNS-3080 (from [17]).

Pinout

Pin	Name	Description
1	NCS	Chip select (active low input)
2	MISO	Serial data output (Master In/Slave Out)
3	SCLK	Serial clock input
4	MOSI	Serial data input (Master Out/Slave In)
5	LED_CTRL	LED control output
6	RESET	Reset input
7	NPD	Power down (active low input)
8	OSC_OUT	Oscillator output
9	GUARD	Oscillator gnd for PCB guard (optional)
10	OSC_IN	Oscillator input
11	NC	No connect
12	OPTP	Connect to VDD3
13	REFC	Reference capacitor
14	REFB	Reference capacitor
15	VDD3	Supply voltage
16	GND	Ground
17	VDD3	Supply voltage
18	NC	No connect
19	GND	Ground
20	NC	No connect

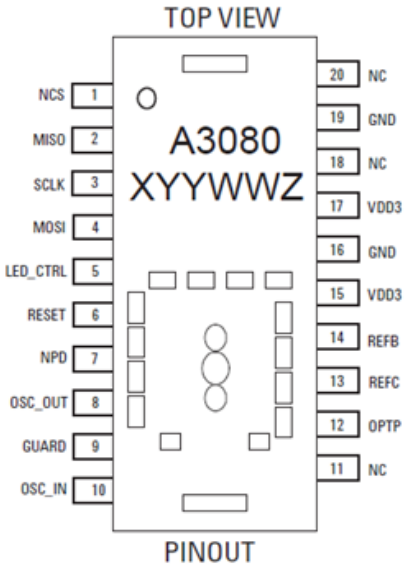


Figure 10. Pinout of ADNS-3080 (from [17]).

The SPI is a synchronous serial port used to read the data from the different registers of the sensor or to select specific parameters such as resolution. To activate the serial connection, NPD must be set to high, RESET to low, and NCS to low. If NCS goes high during a transaction, the transaction is aborted and the SPI is deactivated. The clock input (SCLK) is always generated by the microcontroller. If multiple sensors are

connected to the same master, the NCS can be used to select one sensor and deselect the other. Every register in the ADNS-3080 has a unique address. The different registers and reset values are illustrated in Figure 11.

Address	Register	Read/Write	SROM Default Value
0x00	Product_ID	R	0x17
0x01	Revision_ID	R	0xNN
0x02	Motion	R	0x00
0x03	Delta_X	R	0x00
0x04	Delta_Y	R	0x00
0x05	SQUAL	R	0x00
0x06	Pixel_Sum	R	0x00
0x07	Maximum_Pixel	R	0x00
0x08	Reserved		
0x09	Reserved		
0x0a	Configuration_bits	R/W	0x09
0x0b	Extended_Config	R/W	0x00
0x0c	Data_Out_Lower	R	Any
0x0d	Data_Out_Upper	R	Any
0x0e	Shutter_Lower	R	0x85
0x0f	Shutter_Upper	R	0x00
0x10	Frame_Period_Lower	R	Any
0x11	Frame_Period_Upper	R	Any
0x12	Motion_Clear	W	Any
0x13	Frame_Capture	R/W	0x00
0x14	SROM_Enable	W	0x00
0x15	Reserved		
0x16	Reserved		
0x17	Reserved		
0x18	Reserved		
0x19	Frame_Period_Max_Bound Lower	R/W	0xE0
0x1a	Frame_Period_Max_Bound_Upper	R/W	0x2E
0x1b	Frame_Period_Min_Bound_Lower	R/W	0x7E
0x1c	Frame_Period_Min_Bound_Upper	R/W	0x0E
0x1d	Shutter_Max_Bound_Lower	R/W	0x00
0x1e	Shutter_Max_Bound_Upper	R/W	0x20
0x1f	SROM_ID	R	0x00
0x20-0x3c	Reserved		
0x3d	Observation	R/W	0x00
0x3e	Reserved		
0x3f	Inverse Product ID	R	0xF8
0x40	Pixel_Burst	R	0x00
0x50	Motion_Burst	R	0x00
0x60	SROM_Load	W	Any

Figure 11. ADNS-3080 registers (from [17]).

Understanding write and read operations is crucial to understanding how to exchange data between the microcontroller and the sensor. Write operations consist of two bytes both sent by the master over the master-out-slave-in line (MOSI): one byte for the address and one byte for the data. The first byte containing the address has “1” as its most-significant bit (MSB). The second byte containing the data is read by the sensor on SCLK rising edges. Similarly, read operations consist of two bytes. The address byte has “0” as its MSB and is sent by the microcontroller over MOSI. The data byte is driven by the sensor over the master-in-slave-out line (MISO). The sensor reads MOSI bits on every SCLK rising edge and delivers MISO bits on falling edges of SCLK. Minimum timing between two subsequent operations needs to be respected. The time window to be respected between two back-to-back operations is depicted in Figure 12.

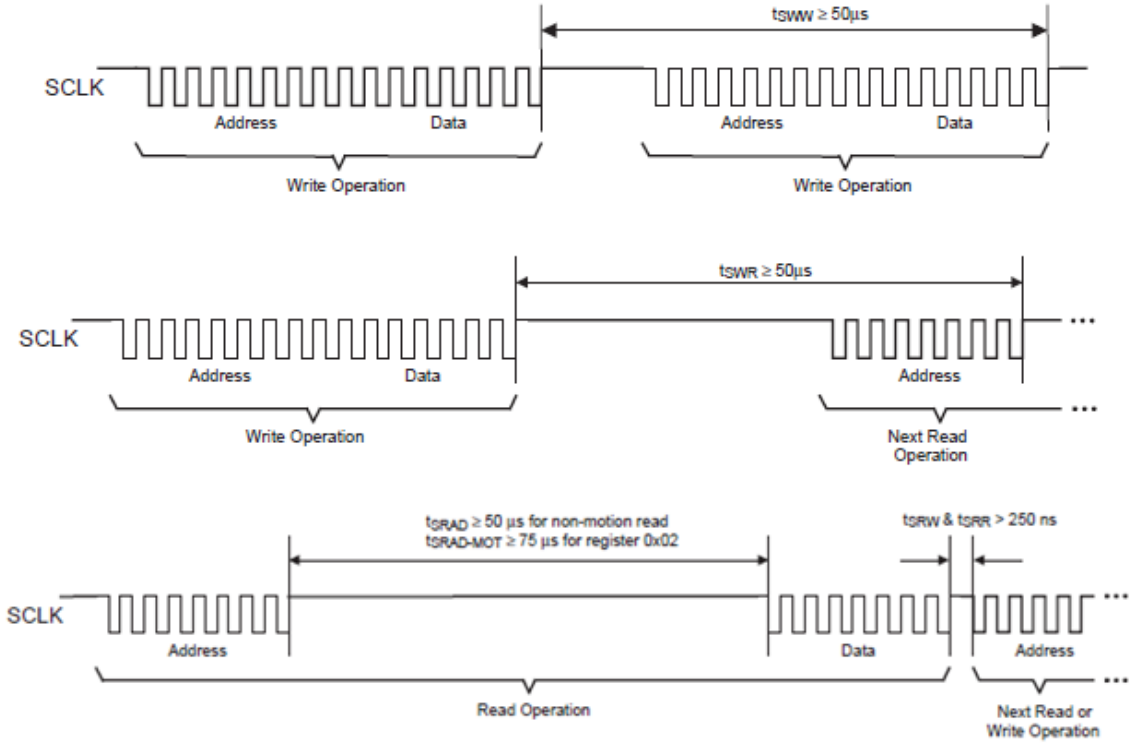


Figure 12. Timing between subsequent operations (from [17]).

The ADNS-3080 has an 8-bit unsigned integer unique ID contained in the “Product_ID” register. The value contained in this register is always the same and is

usually read to make sure that the connection between the microcontroller and the sensor is functional. The “Revision_ID” register can also be used to identify the sensor’s version. Both the “Product_ID” and the “Revision_ID” registers are read only registers. The data, address, and reset values of the “Product_ID” and “Revision_ID” registers are illustrated in Figure 13.

Product_ID **Address: 0x00**

Access: Read Reset Value: 0x17

Bit	7	6	5	4	3	2	1	0
Field	PID_7	PID_6	PID_5	PID_4	PID_3	PID_2	PID_1	PID_0

Data Type: 8-Bit unsigned integer

Revision_ID **Address: 0x01**

Access: Read Reset Value: 0xNN

Bit	7	6	5	4	3	2	1	0
Field	RID_7	RID_6	RID_5	RID_4	RID_3	RID_2	RID_1	RID_0

Data Type: 8-Bit unsigned integer.

Figure 13. “Product_ID” and “Revision_ID” registers (from [17]).

The “Motion” register contains information about the sensor motion and resolution. If motion has occurred since the last time the register was read, the MSB is set to “1”; otherwise, it is set to “0.” The least-significant bit (LSB) allows the user to know the sensor resolution setting. If it is “0,” the resolution is 400 cpi (default value). If it is “1,” the resolution is 1600 cpi. Once a motion has been detected, the user needs to read the “Delta_X” and “Delta_Y” registers to determine the relative displacements Δx and Δy, respectively. The “Motion,” “Delta_X,” and “Delta_Y” registers are all read-only registers. To set the sensor resolution to 1600 cpi, the user needs to access the “Configuration_bits” register and set the RES bit to “1.” The data and reset value of the “Motion,” “Delta_X,” “Delta_Y,” and “Configuration_bits” registers are shown in Figures 14, 15, and 16, respectiveley.

Motion **Address: 0x02**

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	MOT	Reserved	Reserved	OVF	Reserved	Reserved	Reserved	RES

Data Type: Bit field.

Figure 14. "Motion" register (from [17]).

Delta_X **Address: 0x03**

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0

Data Type: Eight bit 2's complement number.

USAGE: X movement is counts since last report. Absolute value is determined by resolution. Reading clears the register.

Delta_Y **Address: 0x04**

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

Data Type: Eight bit 2's complement number.

USAGE: Y movement is counts since last report. Absolute value is determined by resolution. Reading clears the register.

Figure 15. "Delta_X" and "Delta_Y" registers (from [17]).

Configuration_bits **Address: 0x0a**

Access: Read/Write Reset Value: 0x09

Bit	7	6	5	4	3	2	1	0
Field	0	LED_MODE	Sys Test	RES	Reserved	Reserved	Reserved	Reserved

Data Type: Bit field

Figure 16. "Configuration_bits" register (from [17]).

As an optical flow sensor, the ADNS-3080 must be able to operate with a large number of valid features visible in an image or frame; thus, a surface quality "SQUAL" register has to be regularly checked to make sure that the sensor is working properly. The surface quality factor provides an accuracy indication of the relative displacements computed by the sensor. A low "SQUAL" value makes the data collected and computed by the sensor unreliable. The quality factor is directly related to the navigation surface. The "SQUAL" is maximized when the distance between the navigation surface and the imaging lens is optimized. The "SQUAL" register data and reset values are illustrated in Figure 17.

SQUAL **Address: 0x05**

Access: Read Reset Value: 0x00

Bit	7	6	5	4	3	2	1	0
Field	SQ_7	SQ_6	SQ_5	SQ_4	SQ_3	SQ_2	SQ_1	SQ_0

Data Type: Upper 8 bits of a 10-bit unsigned integer.

Figure 17. "SQUAL" register (from [17]).

An imaging lens can significantly increase the maximum tracking speed. It provides a magnification factor equal to the rated tracking speed of the sensor divided by the desired speed. The magnification factor is also given by $\boldsymbol{m=S_i/S_o}$ where $\boldsymbol{S_i}$ is the image distance and $\boldsymbol{S_o}$ is the object distance. From Figure 18, it can be seen that $\boldsymbol{S_i}$ is the distance from the camera to the lens, and $\boldsymbol{S_o}$ is the distance from the surface to the lens.

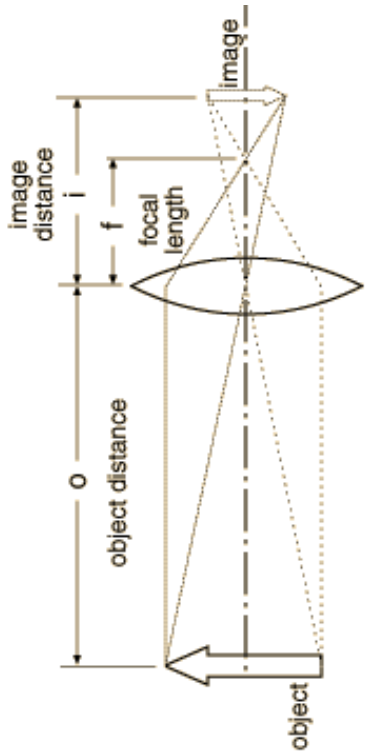


Figure 18. Focal length, object distance, and image distance (from [18]).

In other words, a **1/m** wide surface patch will be focused onto a 1 mm wide image plane within the ADNS-3080. The focal length of the lens can be expressed, according to [19], as

$$\frac{1}{f} = \frac{1}{S_o} + \frac{1}{S_i} = \frac{1}{S_o} + \frac{1}{mS_o} = \frac{1}{S_i / m} + \frac{1}{S_i}. \tag{29}$$

Solving for S_o and S_i, we get

$$S_o = f + \frac{f}{m} = f(1 + \frac{1}{m}) \tag{30}$$

and

$$S_i = f + mf = f(1 + m). \tag{31}$$

By knowing the focal length of an imaging lens, the magnification ratio can be set so that the surface quality factor is maximized and the size of the surface focused onto the camera's image plane increased.

B. DC MOTOR SHIELD

A DC Motor Shield is used to control the speed and the direction of rotation of the wheels mounted to the indoor ground robot. The motor shield uses an H-bridge driver chip L298N integrated circuit that can drive up to two brushed DC motors or a four-wire two phase stepper motor. Each motor can be driven backwards or forwards. The speed of each motor is controlled by high quality, built-in pulse-width Modulated (PWM) signals generated by the microcontroller. The hardware diagram of the DC control board is shown in Figure 19.

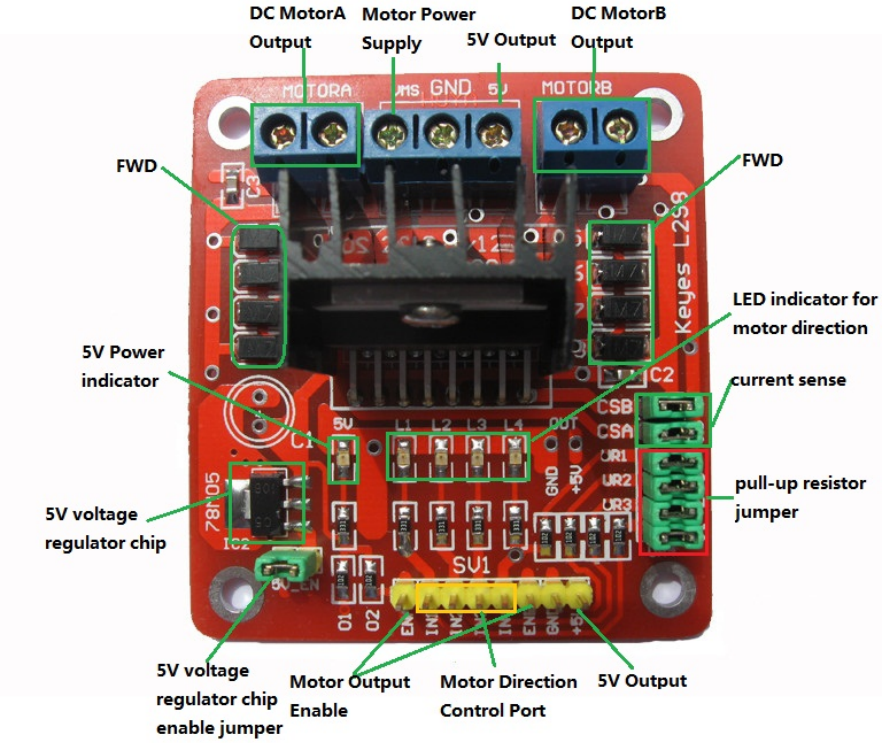


Figure 19. DC Motor Shield parts (from [20]).

To power the board, an external power supply is needed. The input voltage ranges from 6 to 35 volts. All the ports and pins available in the board are listed in Table 5. Motor A is controlled via ports IN1, IN2, and ENA. IN1 and IN2 are used to control the direction of rotation. When IN1 goes high and IN2 goes low, motor A rotates clockwise. On the other hand, when IN1 goes low and IN2 goes high, motor A rotates counter-

clockwise. ENA is connected to a PWM port of the microcontroller to control the speed of the motor. The same applies for motor B on the IN3, IN4, and ENB ports.

Table 5. DC Motor Shield ports.

Port	Description
VMS/GND (inputs)	Power supply pins (6V~35V)
ENA (analog/digital input)	TTL Compatible Enable Input of bridge A (Motor A PWM pin)
IN1 (digital input)	TTL Compatible Inputs of bridge A
IN2 (digital input)	TTL Compatible Inputs of bridge A
ENB (analog/digital input)	TTL Compatible Enable Input of bridge B (Motor B PWM pin)
IN3 (digital input)	TTL Compatible Inputs of bridge B
IN4 (digital input)	TTL Compatible Inputs of bridge B
MOTORA (output)	Output of bridge A
MOTORB (output)	Output of bridge B
5V (output)	5V

C. ARDUINO DUE

Arduino is an open-source physical computing platform that can be used to collect information from several sensors and control a variety of actuators, motors, lights, or other peripherals. All of the collection and control processes are controlled from a single thread of execution in the microcontroller. Arduino provides the user with different microcontroller boards that can be purchased preassembled or as do-it-yourself kits. Arduino is a simplified entry point to create and build digital devices and interactive

objects capable of sensing and controlling the physical world. Arduino also provides the user with an Integrated Development Environment (IDE) that supports C and C++ programming languages. The IDE is a clear and simple programming environment used to write programs for the Arduino board. Also, the IDE gives the opportunity to check the code for possible errors before uploading it to the microcontroller. The Arduino IDE software runs on Windows, Macintosh OSX, and Linux operating systems.

Most of the Arduino boards run at 5.0 volts, except the Arduino Due which runs at 3.3 volts. The ADNS-3080 and the XBEE pro 90 modules both run at 3.3 volts. That means the maximum voltage that the I/O pins can tolerate is 3.3 volts. Forcing the optical mouse sensor or the transceiver module to operate at 5.0 volts can damage them. To avoid any possible incident, the Arduino Due was selected for our work. The Arduino Due front and back sides are displayed in Figure 20.

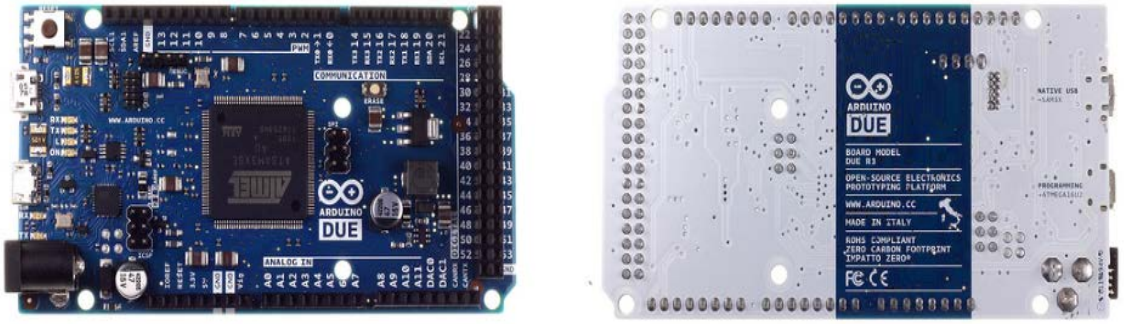


Figure 20. Arduino Due Board (from [21]).

The Arduino Due board contains a 32-bit Atmel SAM3X8E-ARM processor. It has 12 analog inputs and 54 digital input/output pins. Twelve of the digital input/output pins can be used as PWM outputs. The Arduino Due also has a SPI to communicate with another microcontroller or one or more peripheral devices. MISO, MOSI, and SCLK lines are common for all devices. The chip-select or slave-select (SS) line is specific for every device. The microcontroller board operates at 84 MHz clock speed. It has two digital-to-analog converters (DACs), a reset, and an erase button. The Arduino Due pinout is shown in Figure 21.

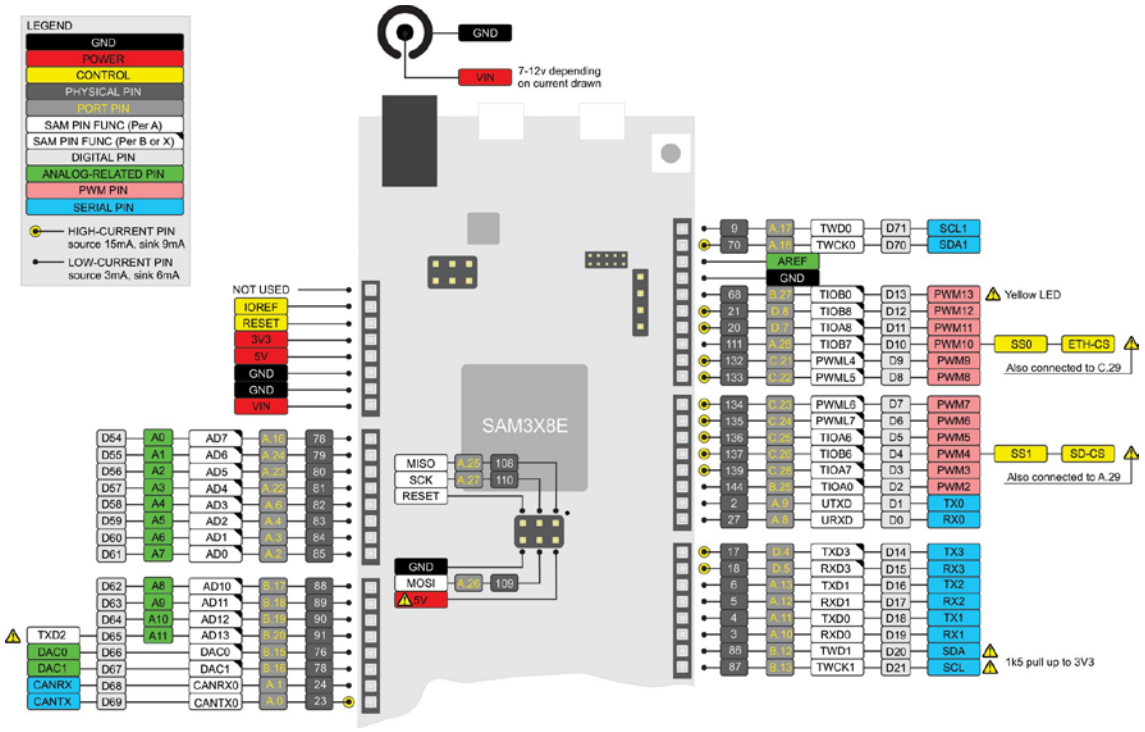


Figure 21. Arduino Due ports (from [21]).

In this work, only one Arduino Due board was used. All the data traffic and control signals were managed by the microcontroller. The Arduino Due is the master and all the other parts of the ground robot (ADNS-3080 modules, XBEE pro 90, DC Motor Shield, and Parallax servo) were slaves.

D. PARALLAX STANDARD SERVO

The Parallax Standard Servo (see Figure 22) is designed to hold any position between 0 and 180 degrees. It is a high precision servo that can be controlled by a microcontroller or device capable of generating PWM signals. From Figure 23, it can be seen that the connection of the servo to any type of microcontroller is easy to realize.

Figure 22. Parallax Standard Servo (from [22]).

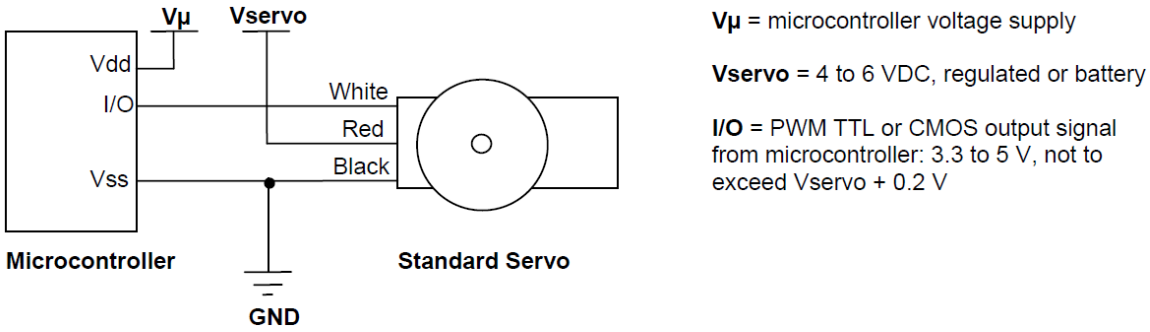


Figure 23. Parallax Standard Servo Wiring Diagram (from [22]).

The position of the servo shaft is directly controlled by the width of the PWM signal pulses. The servo needs a period of 20.0 ms between pulses to hold its position. To center the servo, the microcontroller must deliver a 1.5 ms pulse every 20.0 ms. The PWM signal required for a centered servo is shown in Figure 24.

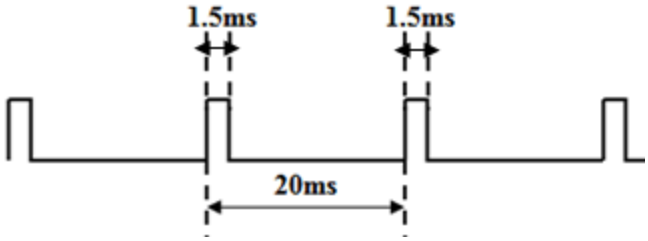


Figure 24. Timing diagram for centered servo.

The pulse duration ranges from 0.75 to 2.25 ms. A 0.75 ms pulse corresponds to the servo shaft positioned at 0 degree. A 2.25 ms pulse corresponds to a servo shaft position of 180 degrees. The center position corresponds to a servo shaft position of 90 degrees. So, depending on the pulse duration, the servo shaft can rotate either clockwise or counter-clockwise.

E. XBEE-PRO 900 DIGIMESH RF MODULES

The XBee-PRO 900 RF modules (see Figure 25) were mainly engineered and designed to be used in wireless sensor networks (WSNs). They are reliable and require low power to operate efficiently. They operate within the ISM (industrial, scientific, and medical) 900-MHz frequency band to support up to 10 km (using high gain antennas) RF line-of-sight ranges and 156 kbps data rates.

Figure 25. XBee-PRO 900 DigiMesh RF module (from [23]).

The XBee-PRO 900 can communicate with any host that has a Universal Asynchronous Receiver/Transmitter (UART) interface. At the source, the UART converts parallel-form data into serial-form data. At the destination, the UART receives the bits of data and reassembles them into bytes of data. Any microcontroller supporting a UART interface can be directly connected to the pins of the RF module. From Figure 26, it can be seen that the UART system data flow diagram is based on a four-wire connection. RTS and CTS pins correspond, respectively, to Request-to-Send and Clear-to-Send pin flow control pins.

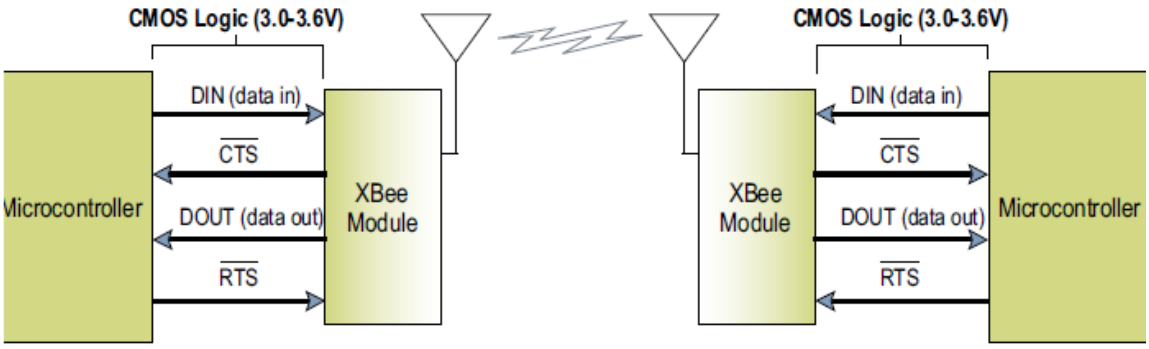


Figure 26. UART Data Flow Diagram (from [23]).

F. TRAJECTORY-FOLLOWER ROBOT

The idea of the trajectory-follower robot came from the principle of operation of the Flight Management System (FMS) in aircraft. The FMS is an embedded system that keeps track of the aircraft's position by collecting data from its various sensors (GPS, INS, radio navigation tools, etc.). The FMS consists essentially of a Flight Management Computer (FMC) connected to the different sensors and a Control Display Unit (CDU). Before take-off, a flight plan is entered by the pilot via the CDU into the FMS. The flight plan is the route the aircraft must follow to fly from the departure point to the destination point and contains all the waypoints needed to reach the destination. Once in flight, knowing the aircraft's position and the flight plan, the FMS can guide the aircraft along the way by controlling the autopilot and the auto-throttle systems. The different parts of a typical FMS are displayed in Figure 27.

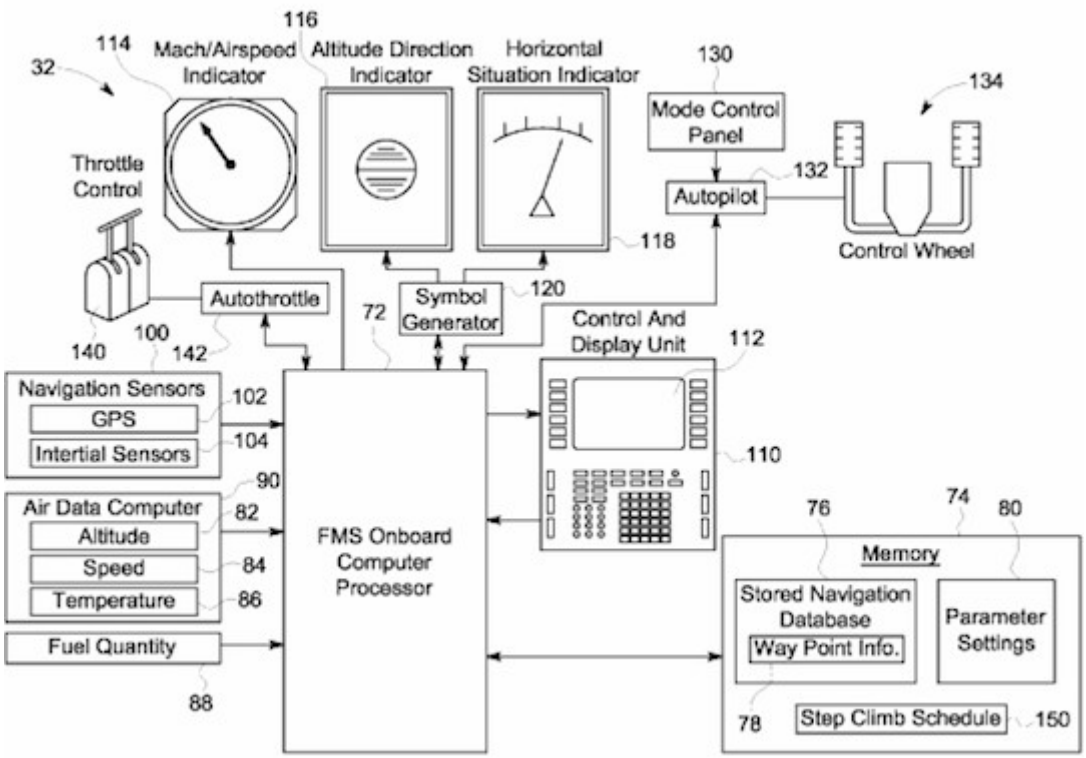


Figure 27. Example of a typical FMS (from [24]).

In this work, the Arduino Due plays the role of the FMC. A desktop computer plays the role of the CDU, and the DC motor board plays the role of the autopilot and auto-throttle systems. Using the Arduino IDE installed in the control unit, we uploaded the program containing the different waypoints to the microcontroller. Once the program was uploaded, the processor collects position information from the ADNS-3080 optical mouse sensor. Given the robot position and route plan, the Arduino Due controls the speed and direction of rotation of the wheels via the DC motor board. The position information and the robot maneuvers are displayed on a display unit located in a monitoring area. The control unit and the display unit can be located at the same place. An RF connection between the display unit and the onboard computer is established by using two XBee-PRO 900 modules. The emitter module is mounted to the robot and directly connected to the microcontroller via one of its four UART interfaces. The receiver module is directly connected to the display unit via the serial port. The different parts of the wheeled-robot's onboard system are shown in Figure 28.

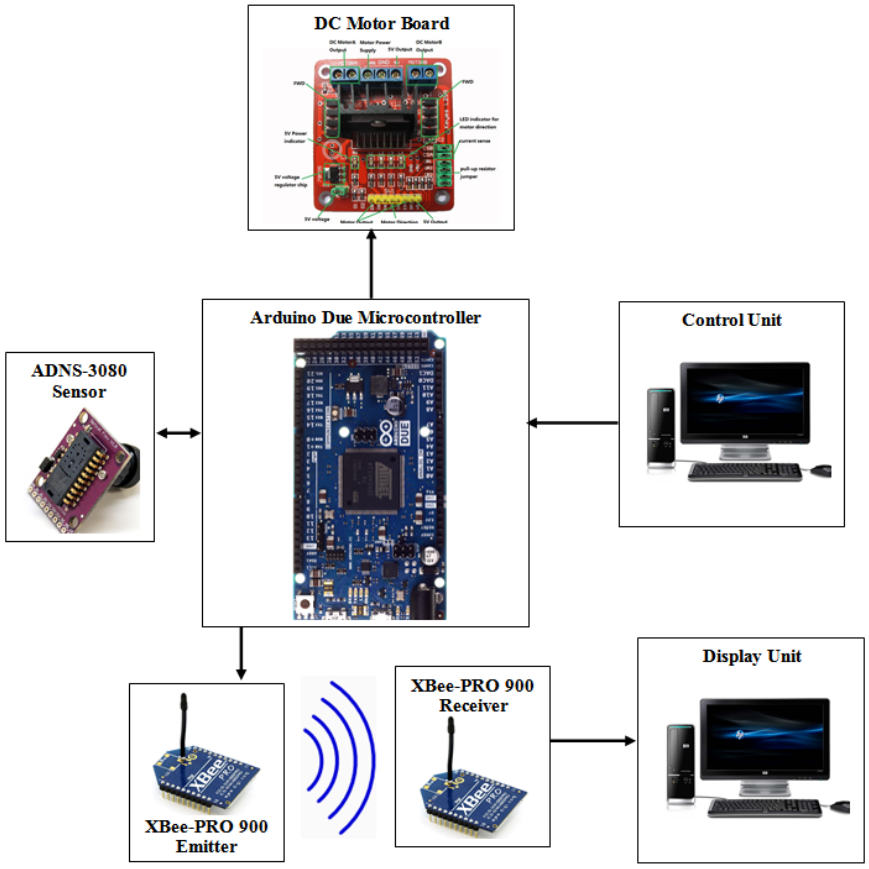


Figure 28. Indoor robot embedded system.

This work involved extensive coding and programming. Every component had to be programmed and tested separately, and all the resulting codes had to be combined into one final program. The resulting program allowed the microcontroller to read and edit the data contained in the different registers of the optical flow sensor, control the speed and direction of rotation for the motors, and continuously send information of position and behavior to the display unit. The master can access all the registers of the ADNS-3080 chip. To track the robot's position, the microcontroller initializes the sensor, configures its settings, and collects position data from all the registers involved in the dead reckoning process. All the useful data are sent out to the display unit via the RF link. Getting a surface quality feedback from the sensor is very important since a high quality factor indicates that the ADNS-3080 can see a large number of terrain features; thus, its

ability to reliably track position improves. A significant drop in the quality factor indicates that the dead reckoning process is no longer reliable.

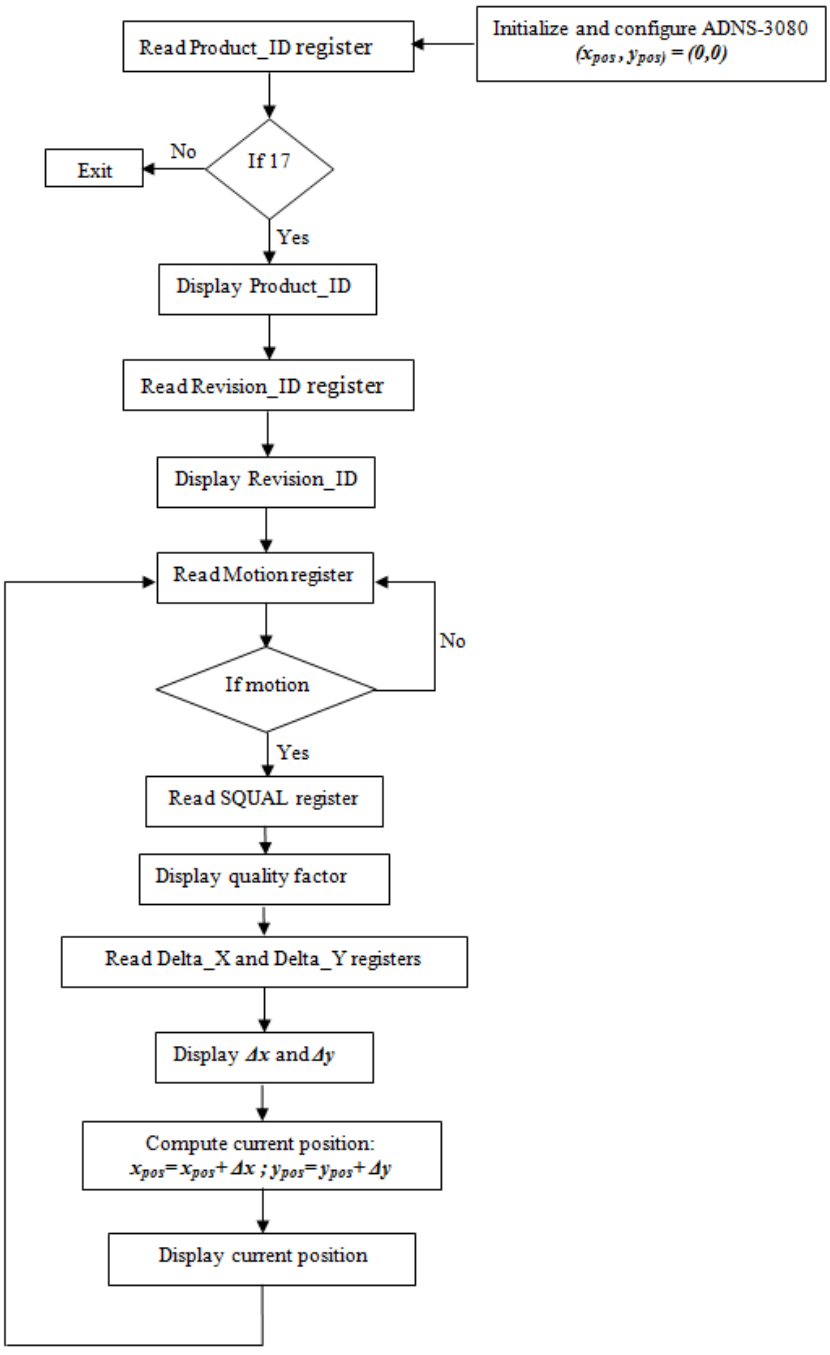


Figure 29. Communication protocol between the Arduino Due and the ADNS-3080 sensor.

The communication process between the microcontroller and the optical mouse sensor is described by the flowchart presented in Figure 29.

As said before, traditional techniques of dead reckoning using data collected and measured by encoders attached to either the robot's wheels or the engine axis suffer from slipping and crawling. Slipping occurs when wheels slide, and crawling occurs when an external force is exerted on the robot. The dead reckoning we propose here is slipping and crawling resistant since the position tracking function is attributed to one ADNS-3080 sensor. The mouse sensor is not bound to any moving part and is capable of reading relative displacements even when an external force is behind the robot movement. Accurate dead reckoning using optical flow sensors usually involves the use of more than one sensor due to the fact that a robot change of direction is hard to measure using only one sensor. In some works, two optical mouse sensors have been used. In others, arrays of optical mouse sensors have been utilized. Indeed, the use of only one optical flow sensor for dead reckoning may seem like a bad idea unless the robot is also equipped with one or multiple other heading sensors. A novel and efficient mean of using one optical mouse sensor pointed at the ground as a heading and dead reckoning sensor is presented in this work.

The first thing we must consider when using a two or four-wheeled robot is how to make a right or a left turn. A common technique is to vary the speed of the wheels. For instance, if the left wheels rotate faster than the right wheels, the vehicle moves to the right, and vice versa. The distance required to make a turn depends directly on how slowly the left (right) wheels spin and how quickly the right (left) wheels spin. With only one optical mouse sensor, this scenario is not appropriate since the robot position changes continuously during the turn; thus, the position tracking process cannot be reliable.

In this work, we opted for another technique to reduce the complexity associated with the above method. In order to make a turn, the right and left wheels rotate at the same speed but in opposite directions. For example, when the right wheels rotate forward and the left wheels rotate backward, the robot makes a left turn. Contrarily, when the right wheels rotate backward and the left wheels rotate forward, the robot makes a right turn. In both cases the robot maintains its current position. Only a change in direction

occurs. In other words, a right or left turn causes only the robot to spin about its Z_R-axis. An example of a right turn is illustrated in Figure 30. As we can see, the position of the robot remains unchanged; however, after the manoeuver, the robot points $\boldsymbol{\theta}$ degrees to the right.

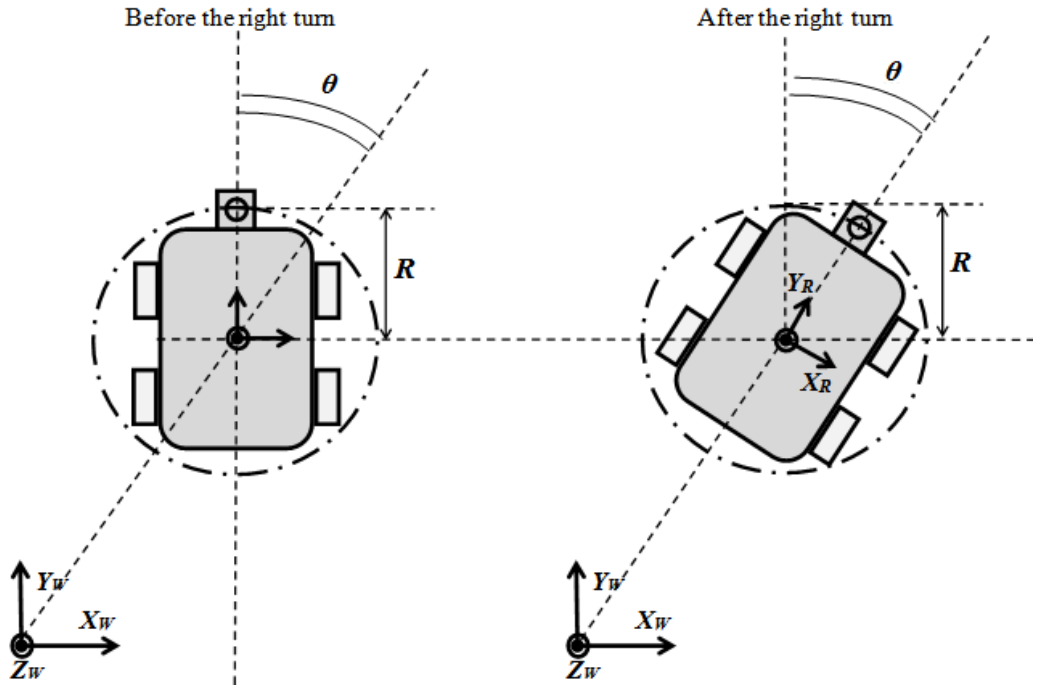


Figure 30. Right turn of $\boldsymbol{\theta}$ degrees.

Similar to an aircraft equipped with an FMS, the trajectory-follower robot needs to move from one point to another. Given the robot's current position and the next waypoint, the onboard computer has to compute the route to follow in order to reach the next destination and then compute the necessary commands to be executed by the motor system. The problem to solve is how to compute and track a change in direction. In this work, after attempting several methods, we determined how to accomplish this using the same sensor, and there is no need to add an additional payload to the platform. When a turn is being executed, only a change in the relative displacement $\boldsymbol{\Delta x}$ is detected. No changes in the relative displacement $\boldsymbol{\Delta y}$ occur. The change relative to the x axis corresponds to an arc of length

$$\left|x_{pos}\right| = \theta \times R\,. \tag{32}$$

For example, a 90-degree right or left turn corresponds to an ***x***-displacement of $\pm \boldsymbol{R \times \pi/2}$. The different phases required to execute a 45 degree right turn are listed in the following flowchart (Figure 31).

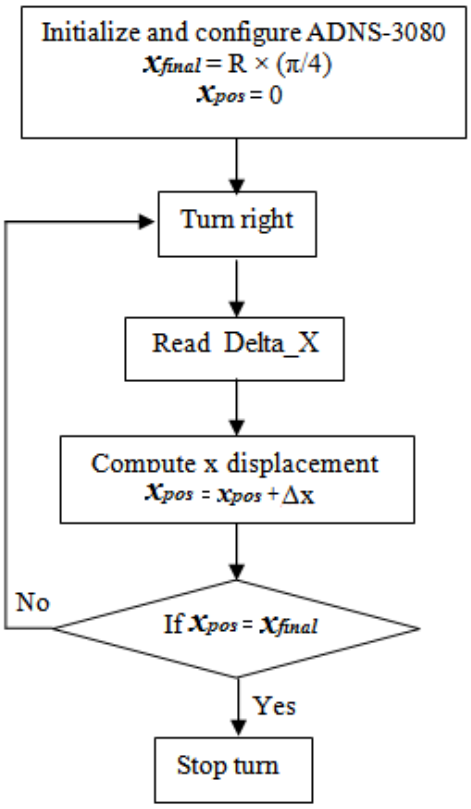


Figure 31. Example of right turn.

The turn speed and direction depend directly on the command signals sent out by the onboard computer to the DC motor board. Given the robot type, the DC motor board can be connected to either two or four DC motors. The wheels are directly mounted to the DC motors. To move from one point to another, the microcontroller needs to first determine the direction and then the distance required to reach the next destination. Every time the robot reaches one waypoint, new computations need to be done. Once done with the computation of direction and distance, the Arduino Due asks the motor system to

translate the data computed into displacement. Meanwhile, the Arduino Due also collects position information from the dead-reckoning sensor. Every time a destination is reached, the robot stops for a period of few seconds to give the microcontroller enough time to calculate the next set of instructions. Consider the example shown in Figure 32. If the robot is to go from point (x_k,y_k) to point (x_{k+1},y_{k+1}), the heading angle and the distance to target are, respectively,

$$\theta = \tan^{-1}\left[\frac{\left|\left(x_{k+1}-x_k\right)\right|}{\left|\left(y_{k+1}-y_k\right)\right|}\right] \tag{33}$$

and

$$D = \sqrt{\left(x_{k+1}-x_k\right)^2+\left(y_{k+1}-y_k\right)^2}\,. \tag{34}$$

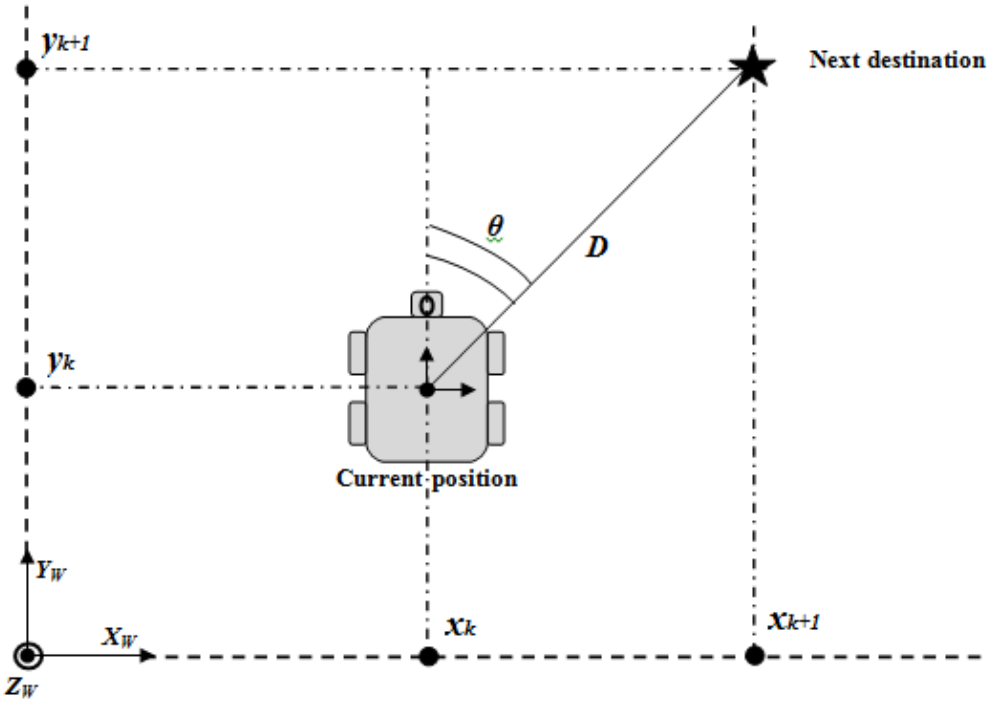


Figure 32. Example of heading and distance to target computation.

The θ and D values calculated above are not valid for all scenarios. Depending on the position of one point with respect to the previous one, we see that the heading angle and distance to target values may vary. From Table 6, we conclude that there are nine possible scenarios.

Table 6. Theta angle and distance to target for all possible scenarios.

Y_W-axis condition	X_W-axis condition	Turn direction	Theta angle	Distance to target
$y_{k+1} > y_k$	$x_{k+1} > x_k$	right	$\theta = \tan^{-1}\left[\frac{\|(x_{k+1} - x_k)\|}{\|(y_{k+1} - y_k)\|}\right]$	$D = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}$
	$x_{k+1} < x_k$	left	$\theta = \tan^{-1}\left[\frac{\|(x_{k+1} - x_k)\|}{\|(y_{k+1} - y_k)\|}\right]$	$D = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}$
	$x_{k+1} = x_k$	No turn	$\theta = 0$	$D = \sqrt{(y_{k+1} - y_k)^2}$
$y_{k+1} < y_k$	$x_{k+1} > x_k$	right	$\theta = \frac{\pi}{2} + \tan^{-1}\left[\frac{\|(y_{k+1} - y_k)\|}{\|(x_{k+1} - x_k)\|}\right]$	$D = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}$
	$x_{k+1} < x_k$	left	$\theta = \frac{\pi}{2} + \tan^{-1}\left[\frac{\|(y_{k+1} - y_k)\|}{\|(x_{k+1} - x_k)\|}\right]$	$D = \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}$
	$x_{k+1} = x_k$	right	$\theta = \pi$	$D = \sqrt{(y_{k+1} - y_k)^2}$
$y_{k+1} = y_k$	$x_{k+1} > x_k$	right	$\theta = \frac{\pi}{2}$	$D = \sqrt{(x_{k+1} - x_k)^2}$
	$x_{k+1} < x_k$	left	$\theta = \frac{\pi}{2}$	$D = \sqrt{(x_{k+1} - x_k)^2}$
	$x_{k+1} = x_k$	No turn	$\theta = 0$	$D = 0$

With all cases addressed for calculating the heading and distance, we now discuss the algorithm adopted and executed by the trajectory-follower robot. At time ***t*=0**, the world frame and the robot frame coincide. All the robot positions are only relative to the world frame. Note that at every waypoint, the robot has to make a θ-turn in the direction opposite to the initial one he made to reach that point. Before every new computation of heading and distance to target, the robot frame *x*- and *y*-axis are, respectively, oriented the same way as the world frame *x*- and *y*-axis. An example of four-waypoint trajectory is illustrated in Figure 33.

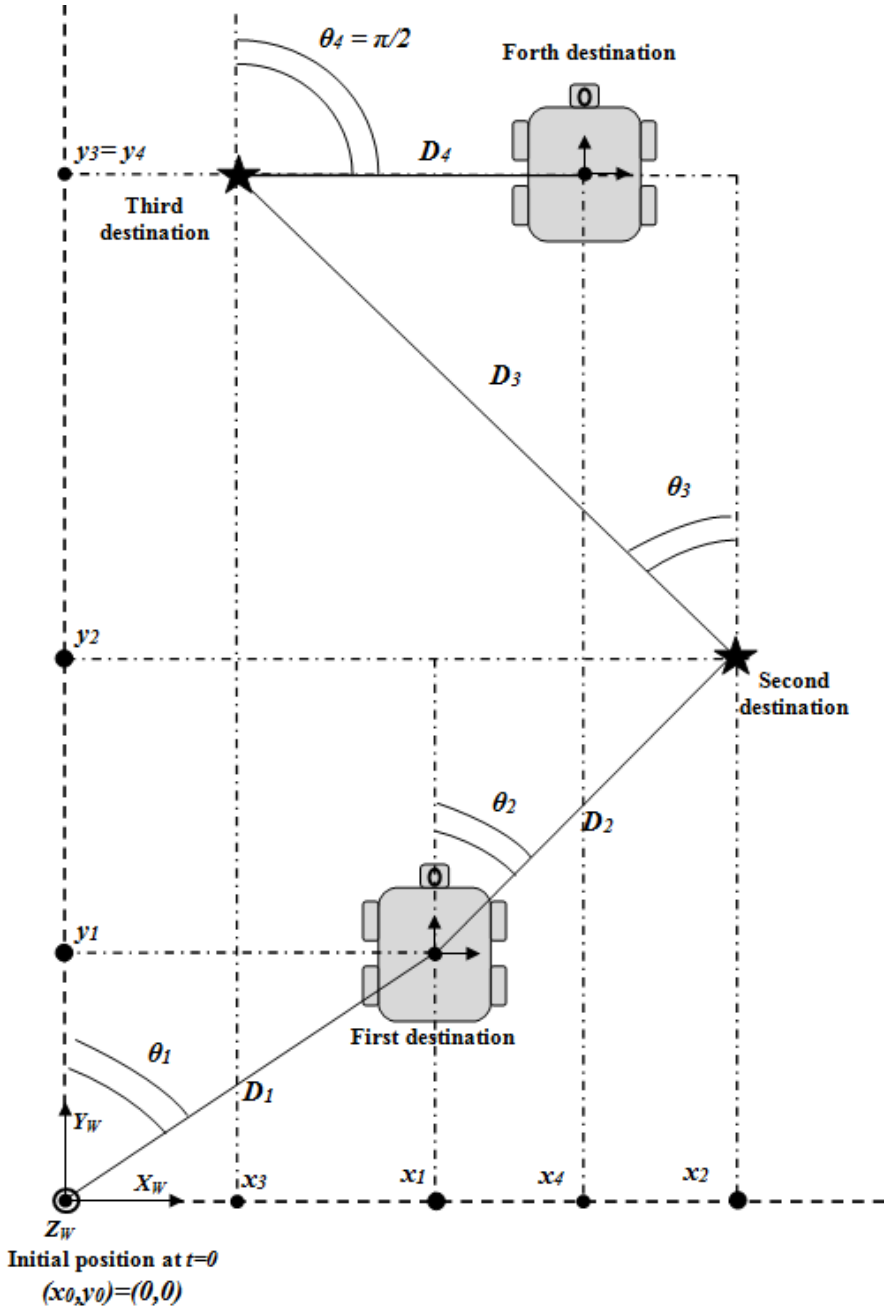


Figure 33. Example of four-waypoint trajectory.

The behavior of the trajectory-follower robot is explained in more detail in the flowchart presented in Figure 34. The flowchart considers all possible scenarios.

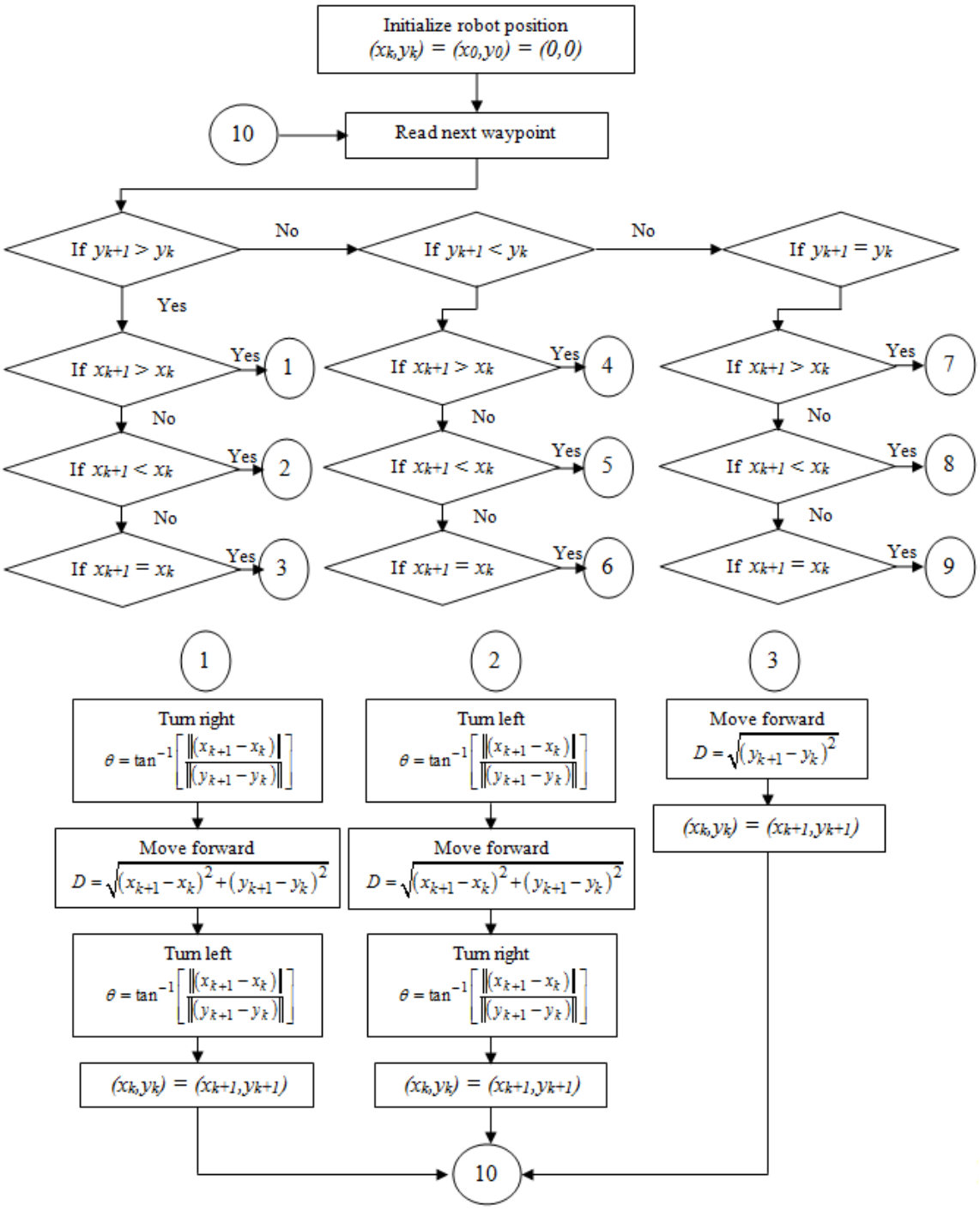


Figure 34. Principle of operation of the trajectory-follower robot (continued on next page).

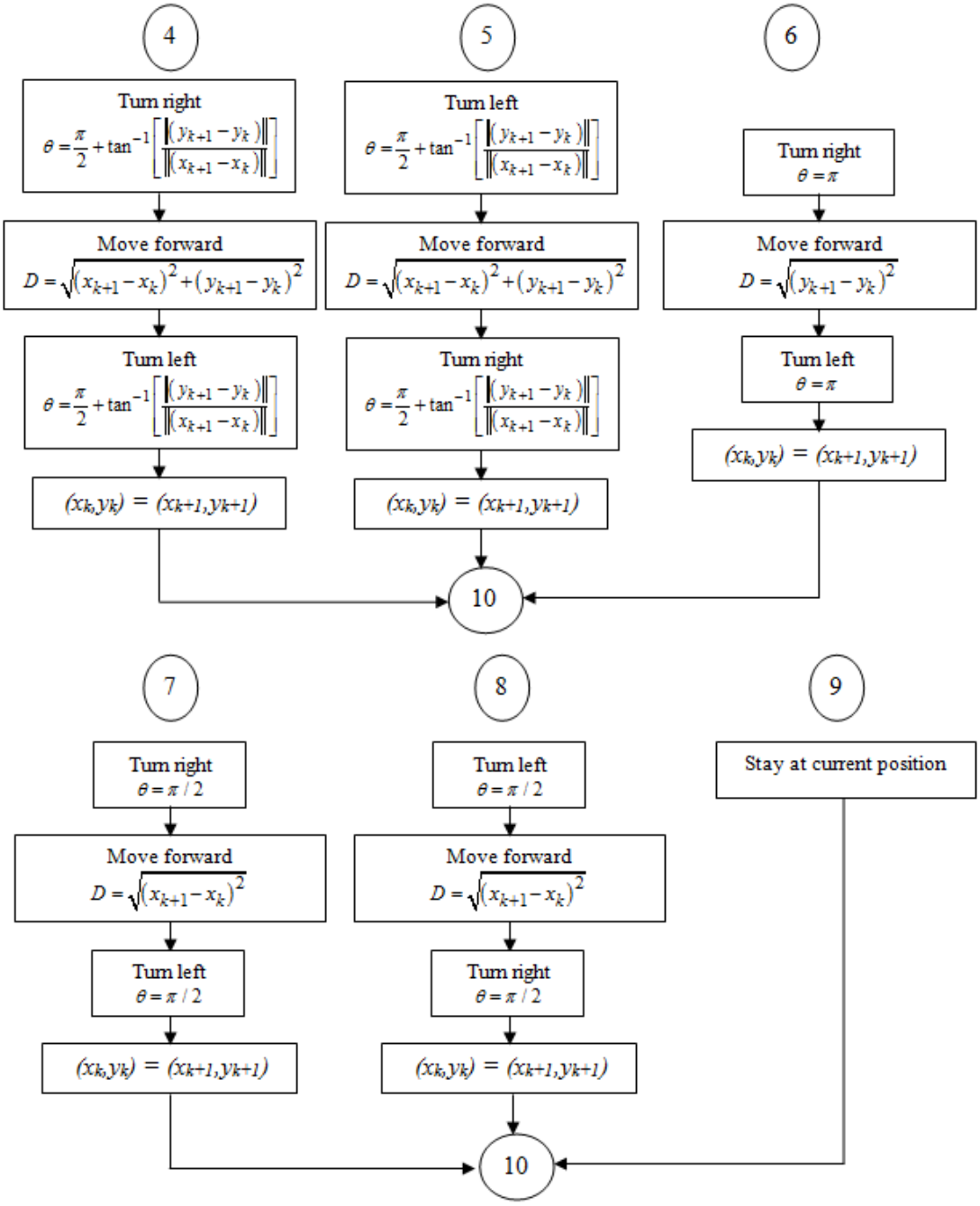


Figure 34. Principle of operation of the trajectory-follower robot (continued from previous page).

G. OBSTACLE DETECTION AND AVOIDANCE ROBOT

Collision avoidance is the set of maneuvers or actions that a robot executes to successfully avoid possible impacts with surrounding objects. An autonomous vehicle cannot avoid collision if its onboard system is unable to detect the presence of objects, so collision avoidance must always be preceded by collision detection. In other words, if a robot is not equipped with collision detection sensors, it is impossible for it to avoid collision with possible obstacles. A hindrance must first be detected and then avoided.

The most widely used sensors for obstacle detection and avoidance are Sound Navigation and Ranging (SONAR) sensors. Active SONARs emit pulses of sounds and listen for possible echoes. The time from emission until the echo is received reflects the distance to the detected object. SONARs are known to be very directional. That means they have a narrow field-of-view. In order for a robot equipped with SONARs to effectively detect and avoid obstacles, more than one sensor is needed. Some of the obstacle avoidance robots available in the market have up to sixteen SONARs to allow 360-degree obstacle detection. At least eight sensors are mounted to the front part of the robot to efficiently avoid frontal collisions. The information collected by SONARs is affected by the direction, orientation, and shape of the obstacle.

The sensor field-of-view is quite important when it comes to obstacle detection and avoidance tasks. If the robot is unable to see enough features, it cannot detect all the surrounding obstacles, so the probability of collision increases. The Hokuyo laser field of view ranges from 240 to 270 degrees. The light detection and ranging (LIDAR) Neato XV-11 is characterized by a 360-degree field of view. The LIDARs seem to be the best solution for obstacle detection and avoidance problems; however, they are expensive and heavy compared to optical flow sensors. All these problems have inspired scientists to investigate optical flow sensor capabilities to replace SONARs and LIDARs.

Optical flow sensors have been used in the last two decades to achieve obstacle detection and avoidance. The majority of existing works are based on the computation of time-to-contact, which requires the implementation of complex algorithms. In this work, we propose a simple, novel, and effective method to perform obstacle detection and

avoidance tasks. Only one ADNS-3080 sensor is used. The main function of the sensor is first to detect possible objects in front of the ground robot and then assist the onboard computer to properly calculate the instructions needed to avoid a possible crash. The set of instructions computed by the microcontroller is executed by the motor system to steer the robot away from the threatening object. To realize this, we used the same platform as in part F but equipped it with an additional ADNS-3080 sensor headed forward. The sensor is mounted on top of an articulated Parallax Standard Servo. The servo is fixed to the upper part of the platform. The locations of the different parts of the robot are shown in Figure 35.

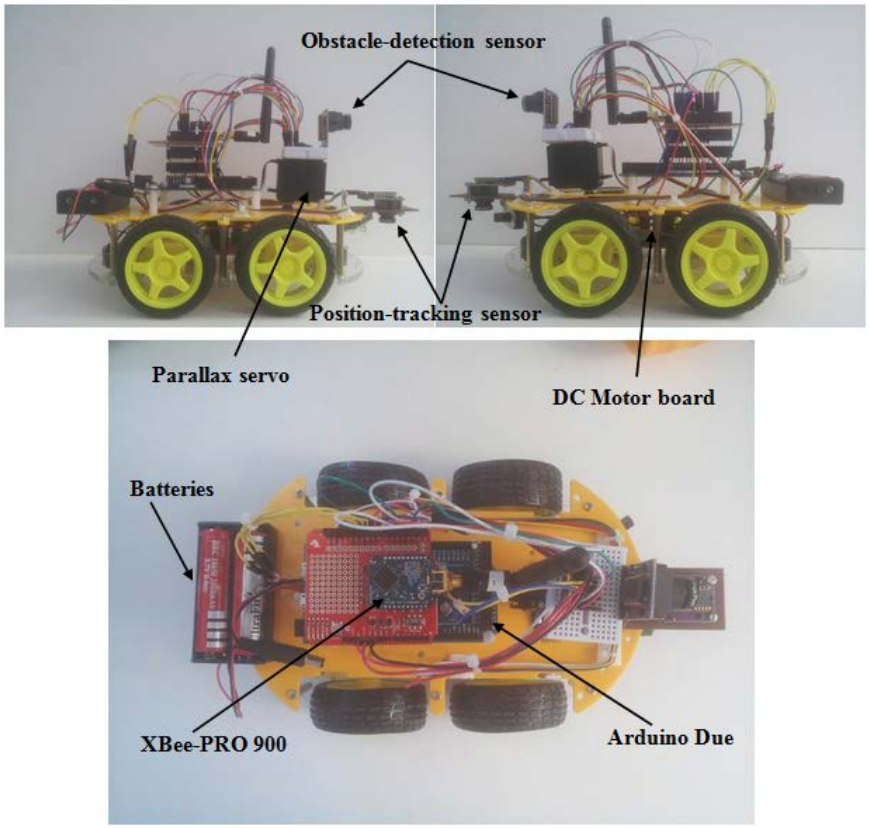


Figure 35. Location of the different parts of the four-wheel robot.

Building a robot that needs to detect and avoid obstacles in order to go from one point to another is a challenging task. In addition to having obstacle detection and avoidance capabilities, such a robot needs to be capable of tracking its own position; thus,

a dead reckoning sensor is required. The good news is that we developed a position tracking sensor (Part F). As a result, we only had to determine how to simultaneously access both sensors such that the robot could execute the obstacle avoidance algorithm while continuously knowing its current position and next direction.

The act of communicating with two sensors at the same time and managing the data coming in and out of the onboard computer was accomplished via the SPI protocol. We said earlier that if multiple sensors have to be connected to a same master, the NCS pin can be used to select one sensor and deselect the other, so the microcontroller can decide which sensor to communicate with by momentarily turning off the NCS input of one sensor and turning on the NCS input of the other one. When position information is needed, the position-tracking data flow must be authorized and the obstacle-detection data flow inhibited, and vice versa.

With the dead reckoning mechanism implemented previously, the only thing remaining is how to detect obstacles. One of the most important registers of the ADNS-3080 is the "SQUAL" register. The data contained in this register reflects how reliable the sensor is. When there is enough light in the room and no obstacle surrounding the sensor, the surface quality factor is large (around 100). On the other hand, as the robot gets closer to an obstacle, this value decreases and drops all the way to zero when the robot is a few centimeters from the object. By closely tracking fluctuations in the surface quality factor, environmental sensing becomes possible. With this collision-detection concept, the microcontroller has to develop and send out to the motor system the set of instructions necessary to avoid crashing into the detected threat. From Figures 36, 37, and 38, it can be seen that depending on how the obstacles are arranged, the robot calls different protocols and mechanisms to avoid collision. As the robot heads towards the destination point, the SQUAL register data varies. When the surface quality factor equals zero, the robot stops for few seconds and checks for the possible existence of obstacles on the right side. If there are no obstacles, the robot makes a 90-degree right turn and then moves forward. Meanwhile, the obstacle-detection sensor is headed toward the obstacle via the Parallax servo. Once the surface quality factor exceeds a specific threshold (50 in this case), the robot makes a 90-degree left turn and moves toward the destination.

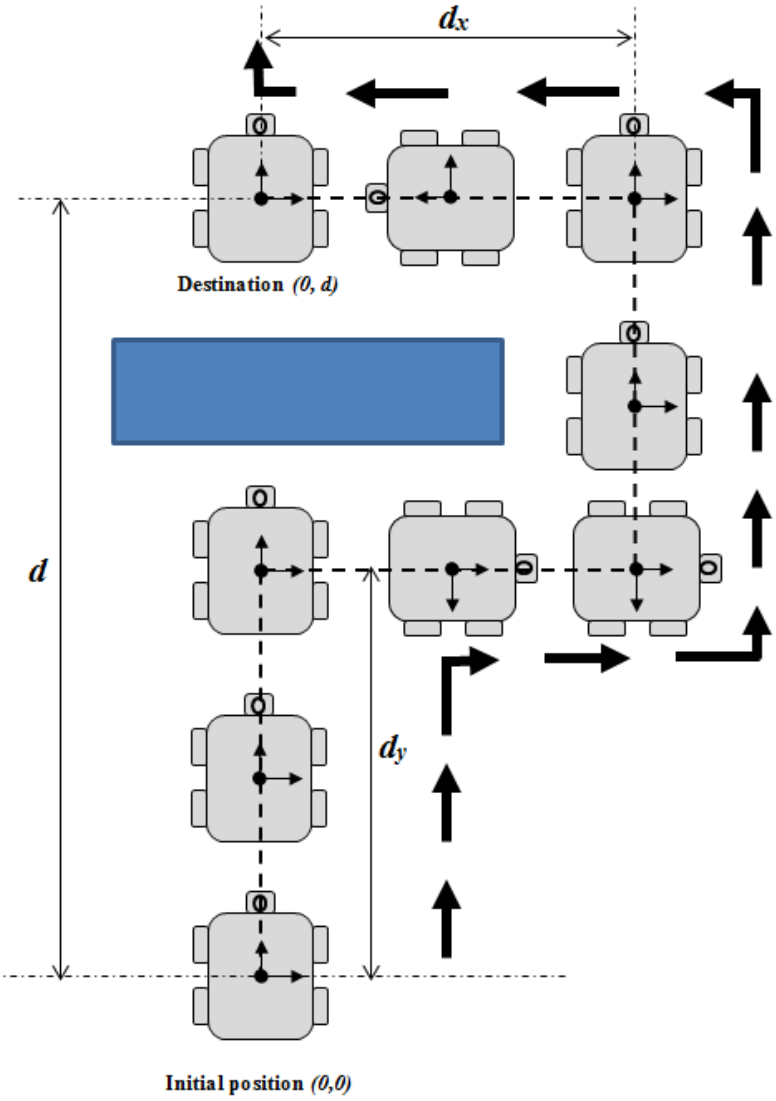


Figure 36. First scenario.

The second scenario involves the existence of obstacles on the right side. The robot checks for the possible existence of obstacles on the left side. If the way is clear, the robot makes a 90-degree left turn and keeps moving forward until the ADNS-3080 detects the obstacle's end. After that, the robot makes a 90-degree right turn and moves towards the destination.

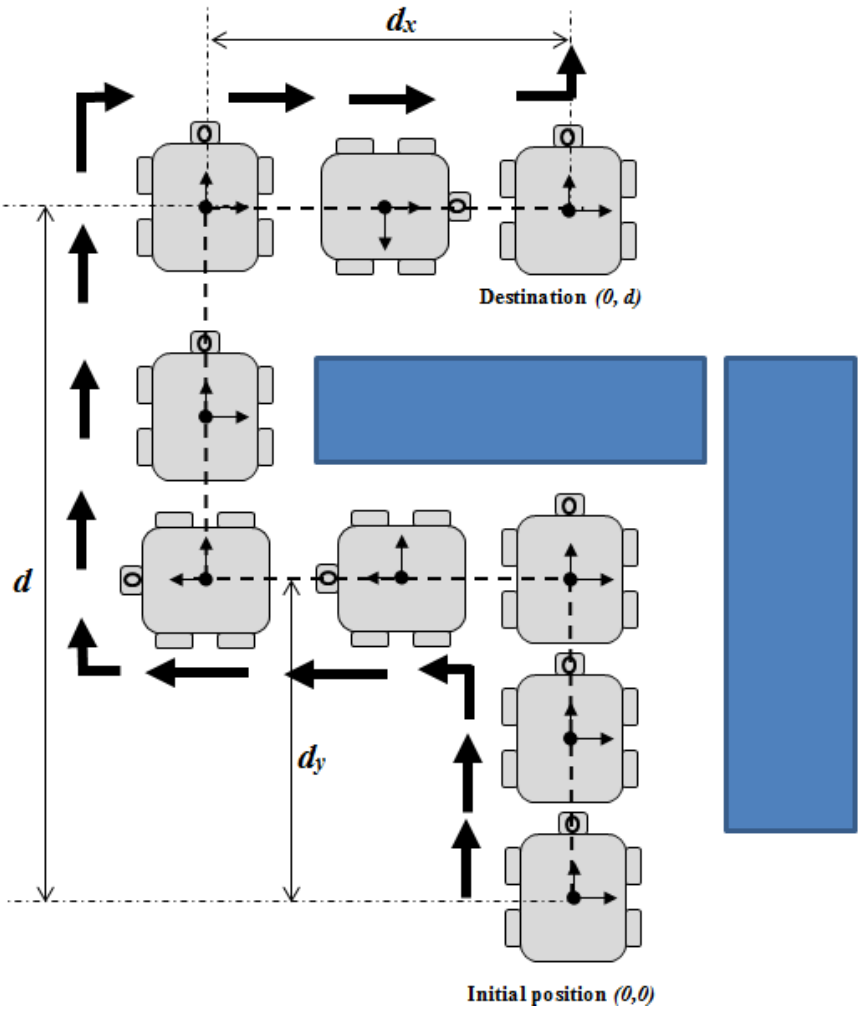


Figure 37. Second scenario.

The worst-case scenario occurs when the robot is trapped in a set of three obstacles. The vehicle has no option but to choose to go one way or the other (right/left). There are more changes of direction and maneuvers to be executed in this case. Here, the detection of the obstacle's end has to be executed twice. The first turn is a 180-degree turn. In all scenarios, the robot changes direction more than once. Without the second ADNS-3080 position-tracking sensor, reaching the desired destination is infeasible. The main advantage of using an optical mouse sensor for collision detection is that it does not care about the shape of the obstacle; however, after several tests, we noticed that the use of dark obstacles leads to better results.

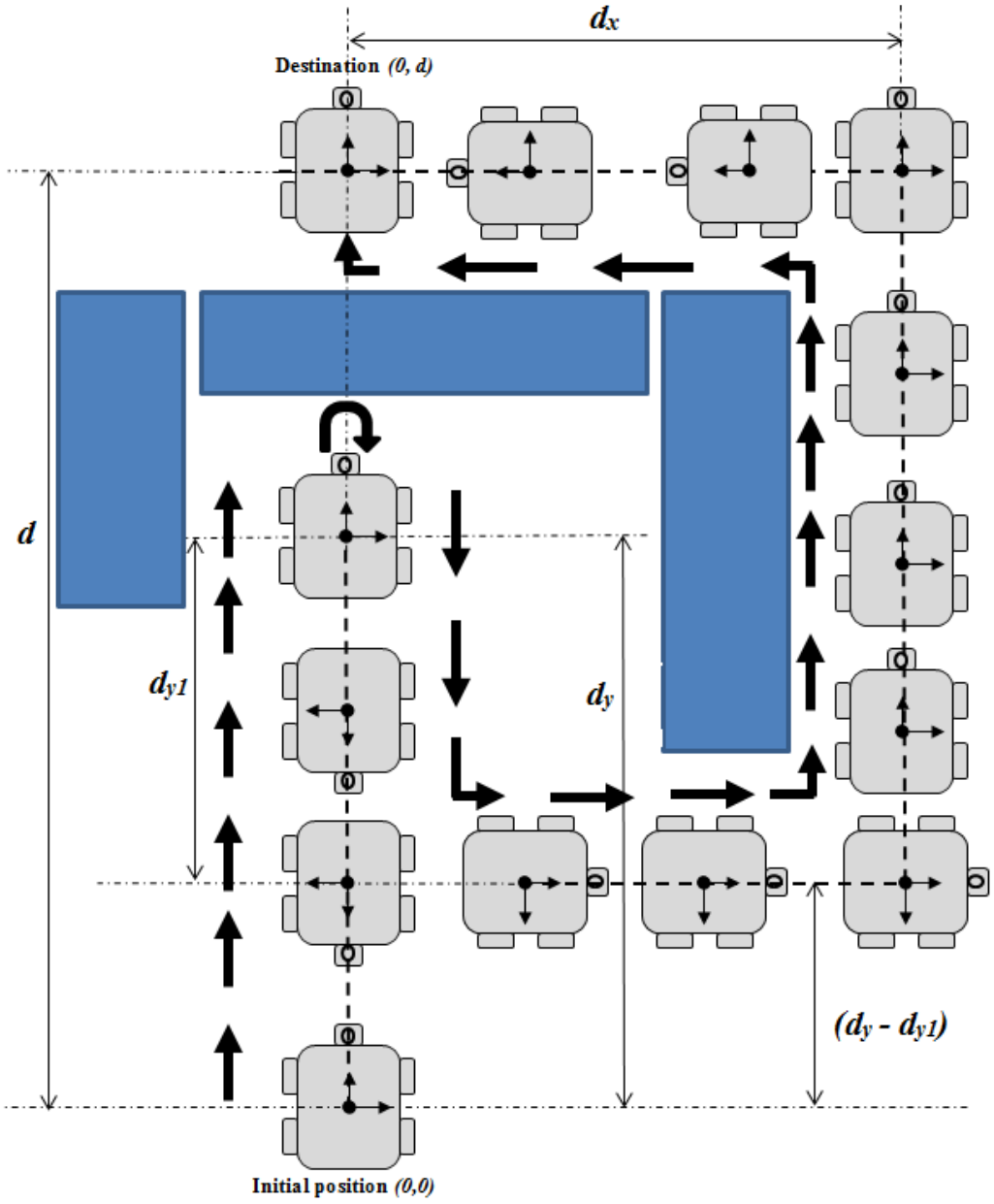


Figure 38. Third scenario.

The principle of operation of the obstacle-detection and avoiding robot is described in detail in the flowchart shown in Figure 39.

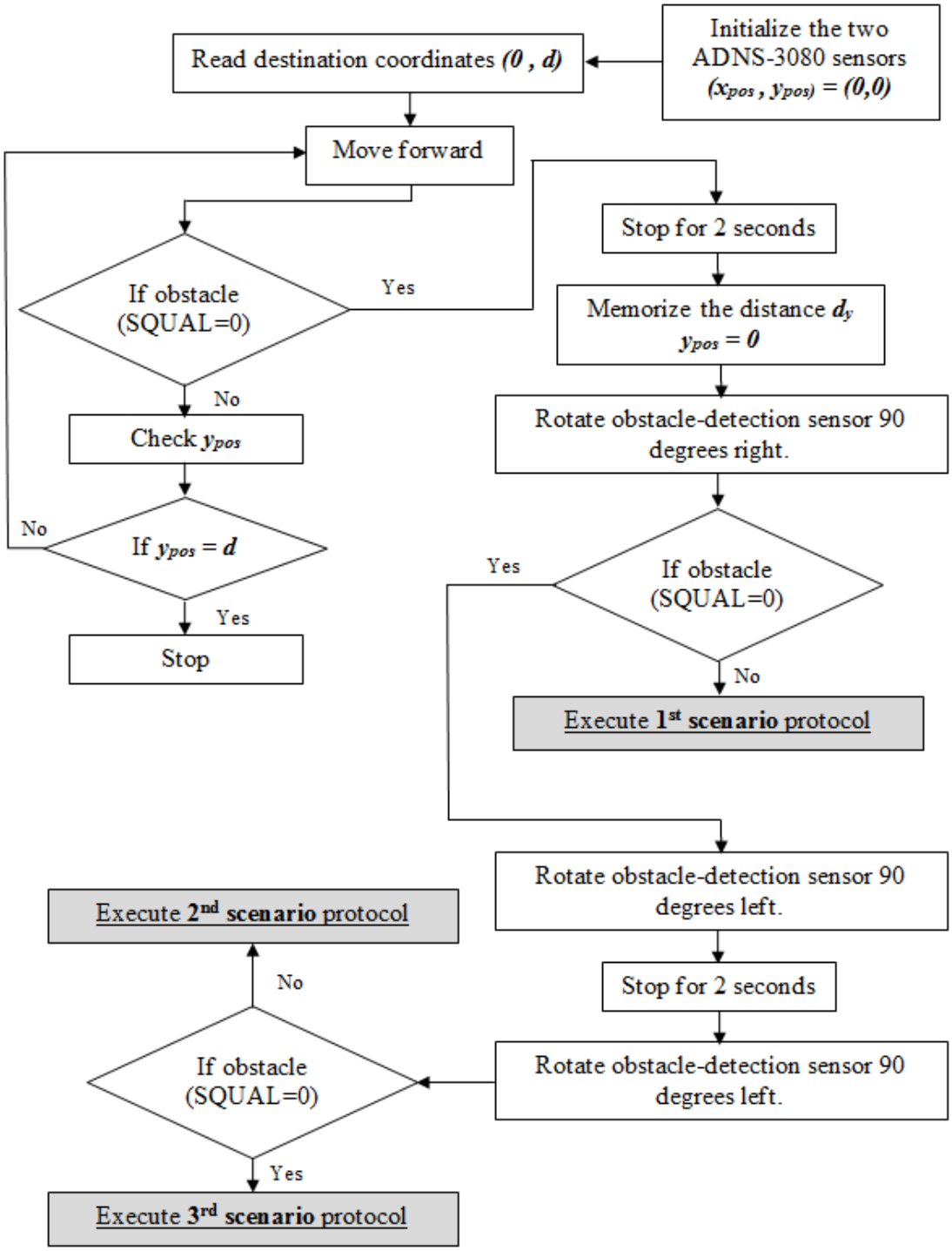


Figure 39. Principle of operation of the obstacle-detection and avoiding robot.

The first, second, and third scenario protocols are depicted, respectively, in Figures 40, 41, and 42.

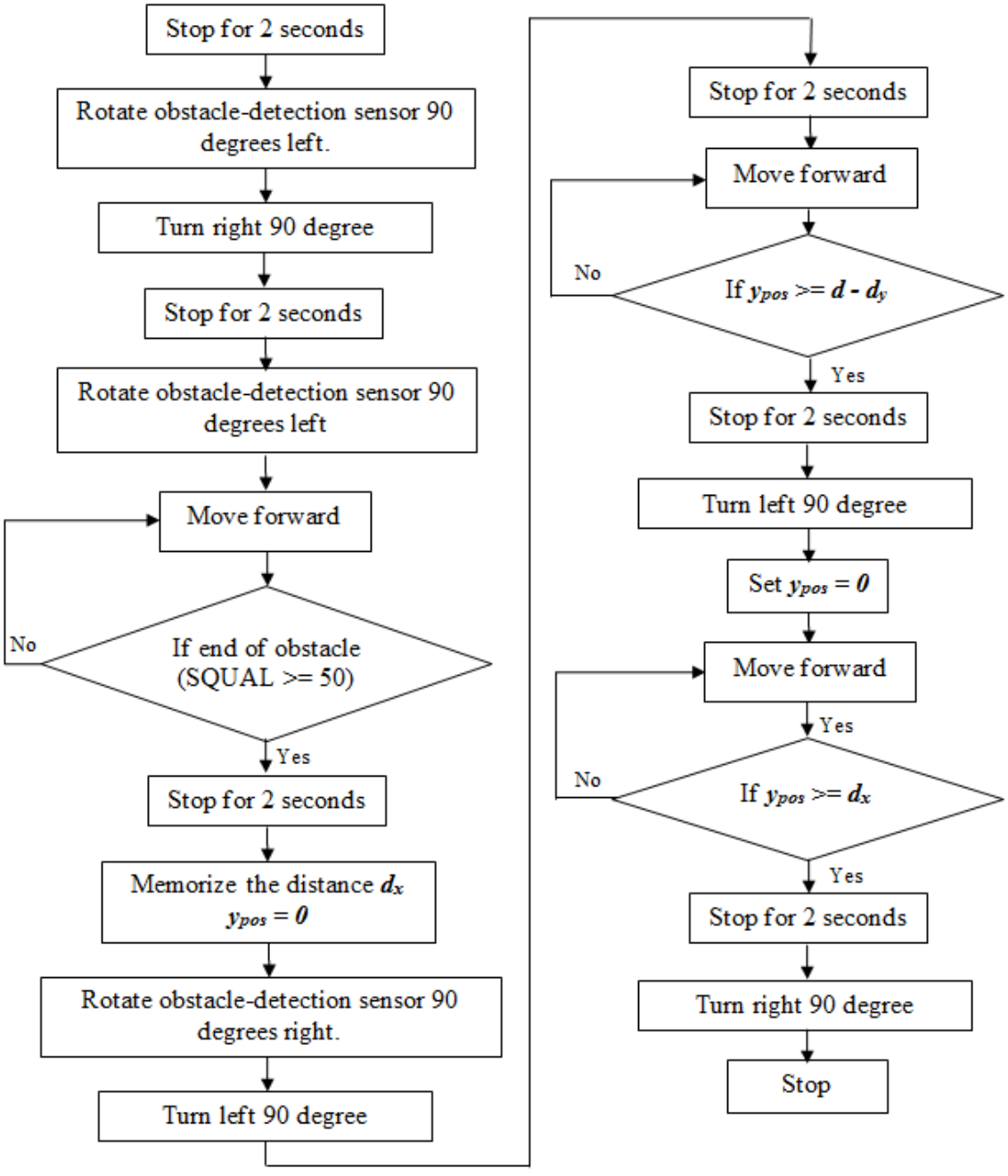


Figure 40. First-scenario protocol.

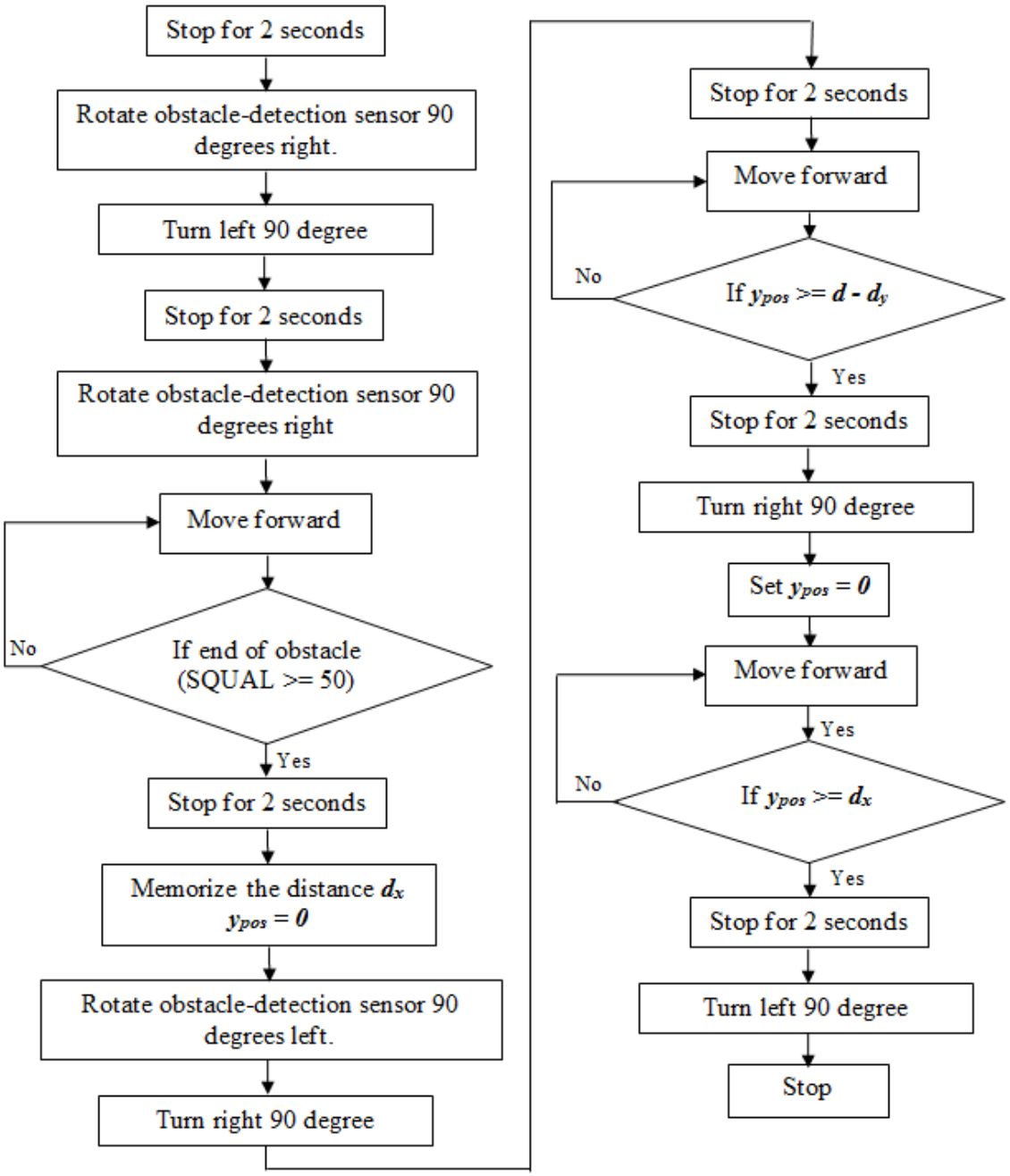
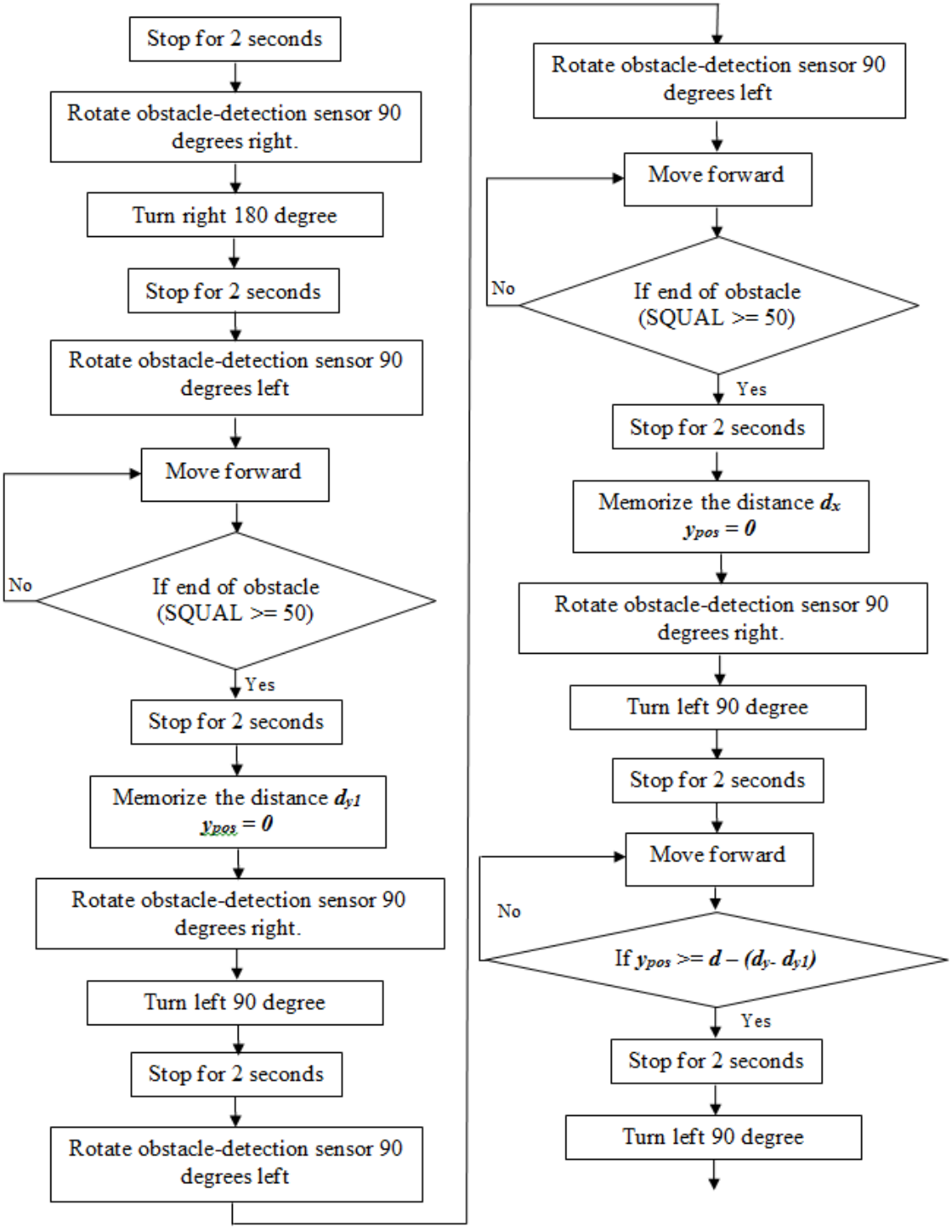


Figure 41. Second-scenario protocol.


Stop for 2 seconds
Rotate obstacle-detection sensor 90 degrees right.
Turn right 180 degree
Stop for 2 seconds
Rotate obstacle-detection sensor 90 degrees left
Move forward
No
If end of obstacle (SQUAL >= 50)
Yes
Stop for 2 seconds
Memorize the distance d_{y1} $y_{pos} = 0$
Rotate obstacle-detection sensor 90 degrees right.
Turn left 90 degree
Stop for 2 seconds
Rotate obstacle-detection sensor 90 degrees left
Rotate obstacle-detection sensor 90 degrees left
Move forward
No
If end of obstacle (SQUAL >= 50)
Yes
Stop for 2 seconds
Memorize the distance d_x $y_{pos} = 0$
Rotate obstacle-detection sensor 90 degrees right.
Turn left 90 degree
Stop for 2 seconds
Move forward
No
If $y_{pos} >= d - (d_y - d_{y1})$
Yes
Stop for 2 seconds
Turn left 90 degree

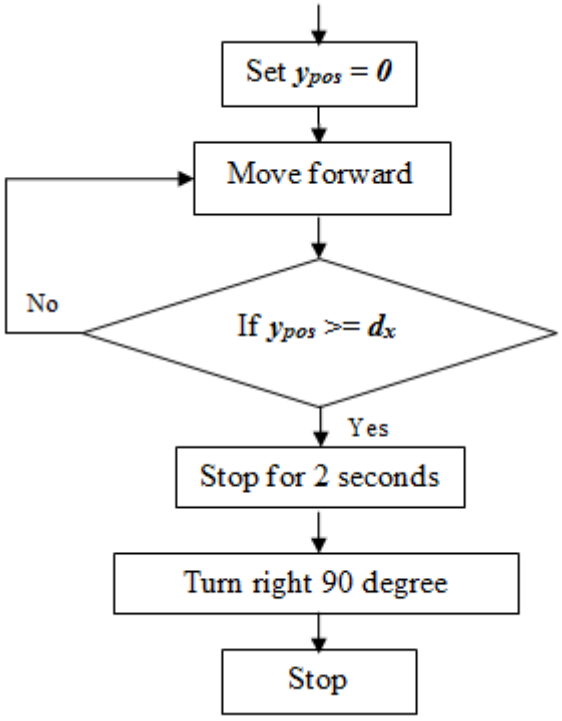


Figure 42. Third-scenario protocol.

When the obstacle's end-detection process is running, the surface-quality-threshold value was set to 50 so that possible objects located far behind the obstacle do not affect the detection's outcome. In this work, we choose for the robot to stop only a few centimeters from the detected obstacle. That explains the choice of the zero-surface-quality value for obstacles detection. We ran other tests for different values of the surface quality factor, and the results were interesting. In order to detect the obstacle and start the obstacle-avoidance process from long distances, a non-zero-surface-quality-factor threshold was selected. That means when the "SQUAL" data is less than the threshold, an obstacle is detected. In that case and considering the three different types of obstacle configurations stated earlier, the detection and avoidance processes are exactly the same. Only one protocol is needed to successfully detect and avoid the different configurations of obstacles. From Figure 43, it can be seen that the robot starts the obstacle-avoidance process without even getting close to the obstacle. The change of the surface-quality-threshold value was accomplished by varying the position of the imaging lens; in other words, varying the object and image distances implemented the desired change. To

summarize, the surface quality factor values for obstacle and obstacle's end detection can be chosen according to the type of application. If the application requires the knowledge of the nature of the obstacles, the first option is recommended. On the other hand, if the application does not care about what kind of obstacle is in the robot's way, the second option might be the most practical since it saves time and makes the robot execute fewer maneuvers.

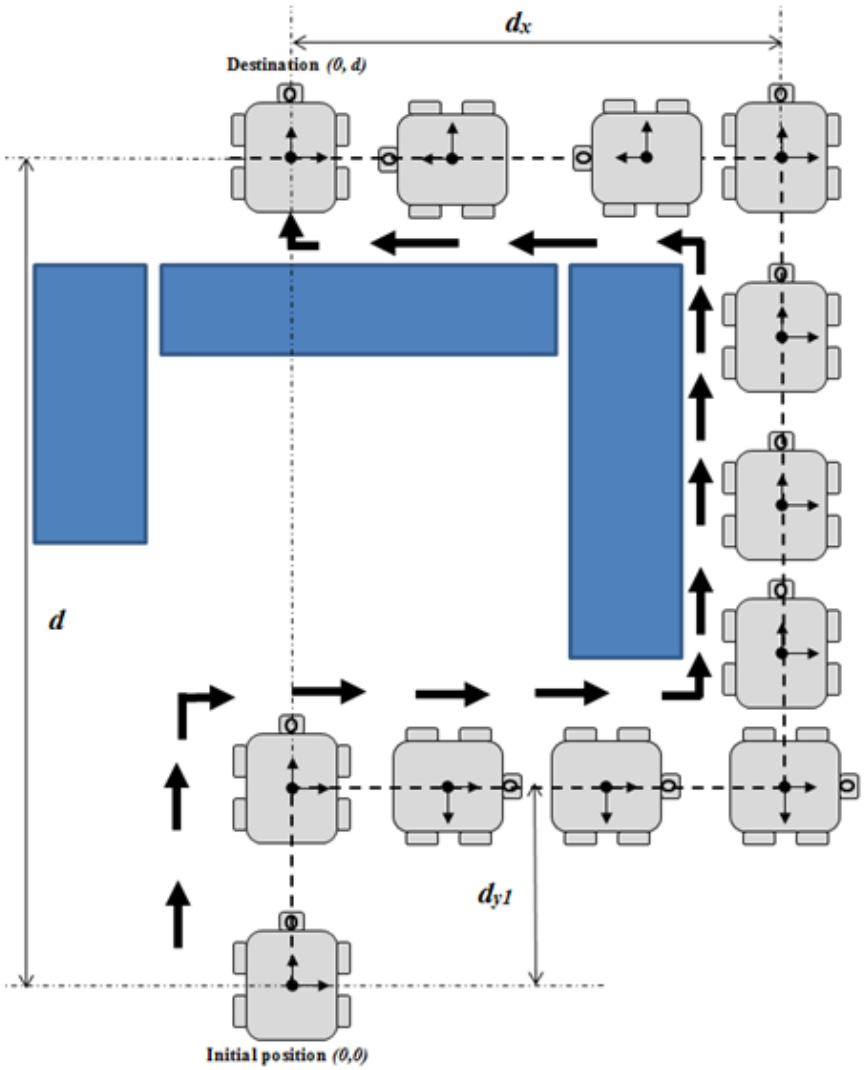


Figure 43. Detection and avoidance protocol for a non-zero-surface-quality-factor threshold.

V. CONCLUSION AND RECOMMENDATIONS

Using two optical mouse chips as position tracking and environment-sensing sensors onboard an indoor-ground robot, we successfully implemented dead reckoning and obstacle-detection and avoidance. The experimental results obtained were promising and can be further improved.

A. RESULTS FOR TRAJECTORY-FOLLOWER ROBOT

The trajectory-follower robot performances proved that an optical flow sensor can be used at the same time as a heading and a relative-displacement calculator. As far as we know, the application of an optical mouse chip as a heading sensor has never been reported before. The originality of this work also resides in the fact that the majority of existing work has used arrays of optical mouse sensors or one optical mouse sensor in association with different other sensors to perform self-localization or dead reckoning, which is not the case here.

B. RESULTS FOR OBSTACLE DETECTION AND AVOIDANCE ROBOT

In robotics, obstacle detection and obstacle avoidance have been investigated by many researchers. Most of the reported works use techniques based on the time-to-contact computation and stereo imaging. These techniques are complex and require more than one vantage point. In this work and for the first time, we proposed a simple and reliable method based on the surface-quality-factor variation. The experimental results obtained were impressive. By using only one optical mouse sensor headed forward, we were able to detect and avoid collisions with obstacles of different shapes and colors. Unlike SONARs, the method we used cannot be affected by the obstacle's shape; however, the experimental tests proved that the results for obstacle detection are better when the obstacle has a dark color.

C. FUTURE WORK

One problem we identified during this work was keeping the robot running in a straight line. Once the robot rotates to the desired heading, the intended path is always a straight line. We adopted the use of an open-loop control system to run the left and right wheels at the same speed. The difference in PWM signals was not very significant, but we were able to drive the robot in a straight line for a few meters. The implementation of a closed-loop control system to solve the problem might be one of the most important recommendations for future works. This can be done by using the relative displacements Δx and Δy as feedback signals returned to the microcontroller. Before computing the duty cycles of the two different PWM signals to be sent to the DC motor board, the microcontroller can check the feedback signals and then decide which wheels need to rotate faster than the others.

The experimental tests proved that the accuracy of the turn was better for angles greater than 20 degrees. In this work, we did not take into consideration the friction force resulting from the nature of the terrain. That is why, from time to time during a turn, the robot's wheels slip. If future work includes the friction parameter in the computation process of turns, the results should improve. Note that the friction force depends directly on the type of surface on which the robot is traversing. The precision in position measurement is highly affected by the variation of the surface quality factor; thus, the results would be more accurate if the robot was running over the same surface during the time of the experiment.

The technique adopted in this thesis can be the subject of future improvement. Optimal control for obstacle detection and avoidance is a very tempting challenge. To get from point A to point B, all the maneuvers executed by the robot to avoid the obstacle were essentially straight-line runs and 90-degree turns; however, with optimal control, the robot deviation from the collision trajectory can be set according to the rate of change in the surface quality. This way, we do not have to wait until the surface-quality factor drops all the way to zero in order to stop the robot and start the obstacle-avoidance procedure; rather, we can trace the rate of degradation in the surface-quality factor and accordingly change the speed of rotation of the left and right wheels to steer the robot

away from the obstacle. An example of how the robot can detect and avoid an obstacle using optimal control is shown in Figure 44. As can be seen, to get from point A to point B, the robot calls fewer mechanisms and executes fewer maneuvers. Once the robot recognizes that it has passed the obstacle, it changes its direction and heads toward point B.

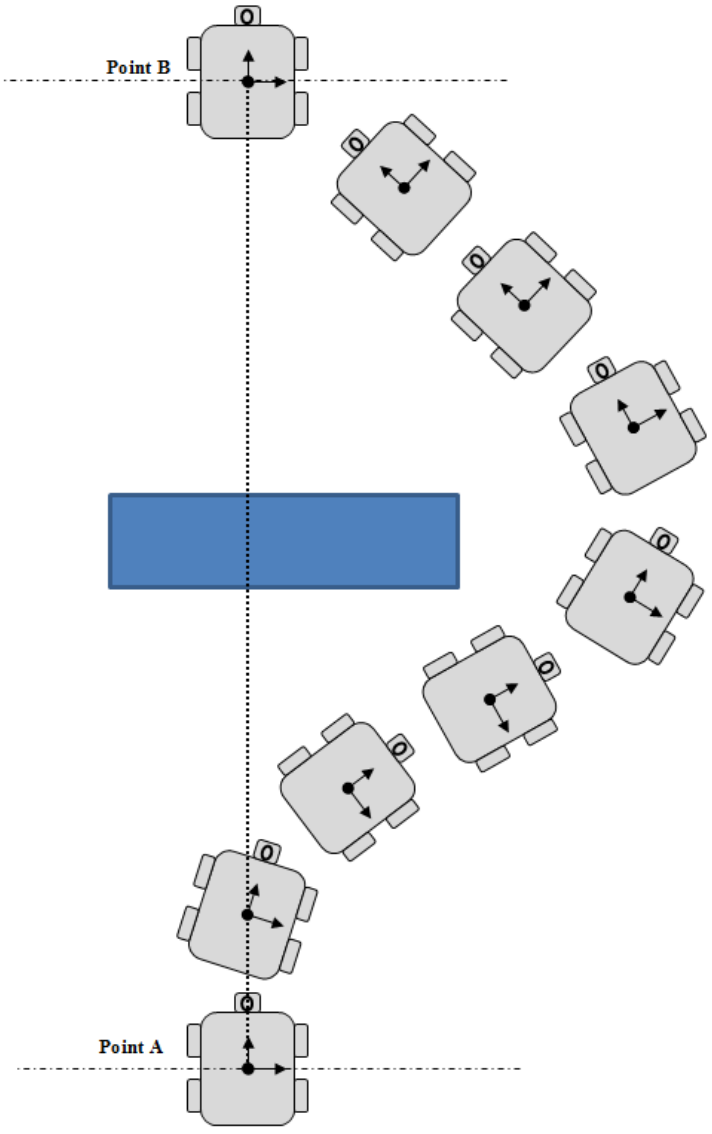


Figure 44. Example of detection and avoidance protocol using optimal control.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. ADNS-3080 SENSOR SCRIPTS

This appendix includes the Arduino scripts written to access the different registers of the ADNS-3080 sensor. These scripts allow the user to check the connection status between the Arduino Due microcontroller and the sensor. Using these scripts, the user can change some of the sensor's default settings.

A. MAIN CODE

```
#include <T3080.h>
#include <SPI.h>
T3080 sensor;
typedef unsigned long Time;
void setup() {
  sensor.init_1();
  Serial.begin(115200);
  Serial.println();
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE3);
  SPI.setClockDivider(SPI_CLOCK_DIV16);
  sensor.reset();
  sensor.init_2();
}
void loop() {
  static Time last_reset;
  Time now = millis();
  sensor.measurement();
  delay(500);
}
```

B. HEADER FILE

```
#ifndef T3080_h
#define T3080_h
#include "Arduino.h"
// Register Map for the ADNS3080 Optical Optical Flow Sensor
#define ADNS3080_PRODUCT_ID            0x00
#define ADNS3080_REVISION_ID           0x01
#define ADNS3080_MOTION                0x02
#define ADNS3080_DELTA_X               0x03
#define ADNS3080_DELTA_Y               0x04
#define ADNS3080_SQUAL                 0x05
#define ADNS3080_PIXEL_SUM             0x06
#define ADNS3080_MAXIMUM_PIXEL         0x07
#define ADNS3080_CONFIGURATION_BITS    0x0a
#define ADNS3080_EXTENDED_CONFIG       0x0b
#define ADNS3080_DATA_OUT_LOWER        0x0c
#define ADNS3080_DATA_OUT_UPPER        0x0d
#define ADNS3080_SHUTTER_LOWER         0x0e
#define ADNS3080_SHUTTER_UPPER         0x0f
#define ADNS3080_FRAME_PERIOD_LOWER    0x10
#define ADNS3080_FRAME_PERIOD_UPPER    0x11
#define ADNS3080_MOTION_CLEAR          0x12
#define ADNS3080_FRAME_CAPTURE         0x13
#define ADNS3080_SROM_ENABLE           0x14
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_LOWER      0x19
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_UPPER      0x1a
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_LOWER      0x1b
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_UPPER      0x1c
#define ADNS3080_SHUTTER_MAX_BOUND_LOWER           0x1e
#define ADNS3080_SHUTTER_MAX_BOUND_UPPER           0x1e
#define ADNS3080_SROM_ID               0x1f
#define ADNS3080_OBSERVATION           0x3d
```

```
#define ADNS3080_INVERSE_PRODUCT_ID    0x3f
#define ADNS3080_PIXEL_BURST           0x40
#define ADNS3080_MOTION_BURST          0x50
#define ADNS3080_SROM_LOAD             0x60
#define FRAME_LENGTH 900
class T3080
{
public:
  T3080() {

       ADNS3080_CHIP_SELECT = 4; // chip select pin
       ADNS3080_RESET = 5; // chip reset pin
       ADNS3080_POWER_DOWN = 6; // Power down pin
     x_pos = 0;
       y_pos = 0;
  }

 void measurement(); // sends values of dx and dy over the serial link
and increments distx and disty respectively by dx and dy
 void reset(); // reset sensor
  byte read_register(byte address);
  void init_1();
  void init_2();
  void write_register(byte address, byte data);

private:
     int ADNS3080_CHIP_SELECT; // chip select pin
       int ADNS3080_RESET;
       int ADNS3080_POWER_DOWN;
     float x_pos;
     float y_pos;
};
#endif
```

C. CPP FILE

```
#include"Arduino.h"
#include"T3080.h"
#include "SPI.h"

void T3080::init_1()
{
    pinMode(ADNS3080_CHIP_SELECT,OUTPUT);
      pinMode(ADNS3080_POWER_DOWN,OUTPUT);
      pinMode(ADNS3080_RESET,OUTPUT);
      digitalWrite(ADNS3080_CHIP_SELECT,LOW);
      digitalWrite(ADNS3080_RESET, LOW);
      digitalWrite(ADNS3080_POWER_DOWN,HIGH);
}

void T3080::init_2()
{
  int retry = 0;
  byte productId = 0;
  byte revisionId = 0;

 // PRODUCT ID VERIFICATION
  while( retry < 10 ) {
    delayMicroseconds(75);
    // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT,LOW);
    // send the device the register you want to read:
    SPI.transfer(ADNS3080_PRODUCT_ID);
      // small delay
      delayMicroseconds(75);
      // send a value of 0 to read the first byte returned:
    productId = SPI.transfer(0x00);
```

```
            if( productId == 0x17 ) {
              Serial.println("\n Found productId ");
                Serial.print(productId, HEX);;
            }

            else{
            Serial.println("\n False productId ");
            Serial.print(productId, HEX);;
            }
          retry++;
           }
   if(productId != 0x17) {
   delay(100);
   exit(1);
    }

// REVISION ID VERIFICATION
   delayMicroseconds(75);
   // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT,LOW);
    // send the device the register you want to read:
    SPI.transfer(ADNS3080_REVISION_ID);
      // small delay
      delayMicroseconds(75);
      // send a value of 0 to read the first byte returned:
    revisionId = SPI.transfer(0x00);
    Serial.println("\n rev");
    Serial.print(revisionId, HEX);

// Set resolution to 1600 counts/inch
    delayMicroseconds(75);
      // set the chip select to low to select the device
```

```
    digitalWrite(ADNS3080_CHIP_SELECT, LOW);

      // send register address
    SPI.transfer(ADNS3080_CONFIGURATION_BITS | 0x80 );

      // send data
      SPI.transfer(0x10);
}

byte T3080::read_register(byte address)
{
    byte data = 0;
    delayMicroseconds(75);

   // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT,LOW);

    // send the device the register you want to read:
    SPI.transfer(address);

      // small delay
      delayMicroseconds(75);

      // send a value of 0 to read the first byte returned:
    data = SPI.transfer(0x00);

    // take the chip select high to de-select:
    digitalWrite(ADNS3080_CHIP_SELECT, HIGH);

      return data;
}

void T3080::write_register(byte address, byte data)
```

```
{
      delayMicroseconds(75);

      // set the chip select to low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT, LOW);

      // send register address
    SPI.transfer(address | 0x80 );

      // send data
      SPI.transfer(data);

    // set the chip select to high to de-select the device
    digitalWrite(ADNS3080_CHIP_SELECT, HIGH);

}

 void T3080::measurement()
{
    byte motion = 0;
      uint8_t dx = 0;
      uint8_t dy = 0;
      int8_t delta_x = 0;
      int8_t delta_y = 0;
      float DELTA_x = 0;
      float DELTA_y = 0;
      byte SQUAL = 0;

      delayMicroseconds(75);
      // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT,LOW);
    // send the device the register you want to read:
    SPI.transfer(ADNS3080_MOTION);
```

```
    // small delay
    delayMicroseconds(75);
    // send a value of 0 to read the first byte returned:
  motion = SPI.transfer(0x00);

  delayMicroseconds(75);
    // send the device the register you want to read:
  SPI.transfer(ADNS3080_DELTA_X);
    // small delay
    delayMicroseconds(75);
    // send a value of 0 to read the first byte returned:
  dx = SPI.transfer(0x00);

    delayMicroseconds(75);
    // send the device the register you want to read:
  SPI.transfer(ADNS3080_DELTA_Y);
    // small delay
    delayMicroseconds(75);
    // send a value of 0 to read the first byte returned:
  dy = SPI.transfer(0x00);

    delayMicroseconds(75);
    // send the device the register you want to read:
  SPI.transfer(ADNS3080_SQUAL);
    // small delay
    delayMicroseconds(75);
    // send a value of 0 to read the first byte returned:
  SQUAL = SPI.transfer(0x00);

// check for overflow
    if( (motion & 0x10) != 0 )
    {
          Serial.println("\n Attention Overflow");
```

```
    }
    else
    {
        Serial.println("\n No Overflow");
    }

    // check resolution
    if( (motion & 0x01) != 0 )
    {
          Serial.println("\n Resolution = 1600 counts/inch");
    }
    else
    {
          Serial.println("\n Resolution = 400 counts/inch");
    }

    // check for motion and update dx and dy
    if( (motion & 0x80) != 0 )
    {
          Serial.println("\n Motion");
    }
    else
    {
          Serial.println("\n No Motion");
    }

delta_x= (int8_t)dx;
delta_y= (int8_t)dy;

DELTA_x=(float)delta_x/1600.0;
DELTA_y=(float)delta_y/1600.0;

DELTA_x=DELTA_x/1.62914206;
```

```
  DELTA_y=DELTA_y/1.62914206;

  DELTA_x=100*DELTA_x;
  DELTA_y=100*DELTA_y;

  x_pos=x_pos+DELTA_x; // x position in cm
  y_pos=y_pos+DELTA_y; // y position in cm
  //Display DELTA
  Serial.print(DELTA_x, DEC);
  Serial.print("                              ");
  Serial.print(DELTA_y, DEC);
  //Display x and y
  Serial.print("                              ");
  Serial.print(x_pos, DEC);
  Serial.print("                              ");
  Serial.println(y_pos, DEC);
  //Display Quality
  Serial.println("\n Quality");
  Serial.print(SQUAL, DEC);

  // take the chip select high to de-select:
  // digitalWrite(ADNS3080_CHIP_SELECT, HIGH);
  //  delayMicroseconds(5);
}

void T3080::reset()
{
    digitalWrite(ADNS3080_RESET,HIGH);                // reset sensor
      delayMicroseconds(10);
      digitalWrite(ADNS3080_RESET,LOW);                       // return
sensor to normal
}
```

D. KEYWORDS FILE

```
T3080       KEYWORD1
measurement KEYWORD2
reset       KEYWORD2
read_register     KEYWORD2
write_register    KEYWORD2
init_1      KEYWORD2
init_2      KEYWORD2
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. DC MOTORS SCRIPTS

The Arduino scripts contained in this appendix allow the user to test the four DC motors. Using these scripts, the user will be able to control the speed and direction of rotation of the right and left wheels.

A. MAIN CODE

```
#include <Motors.h>
#include <SPI.h>
Motors Motion;

void setup()
{
  Motion.initialize();
  Motion.forw();

  Motion.speed_up();       //fade in from 0-255
  Motion.forward();      //continue full speed forward
  delay(2000);
  Motion.slow_down();      //Fade out from 255-0
  Motion.stop();
  delay(2000);           // stop for 2 seconds


  Motion.back();

  Motion.speed_up();       //fade in from 0-255
  Motion.backward();     //full speed backward
  delay(2000);
  Motion.slow_down();      //Fade out from 255-0
  Motion.stop();
  delay(2000);           // stop for 2 seconds
```

```
  Motion.forw();
  Motion.forw_right();   // turn right
  delay(3000);
  Motion.stop();
  delay(2000);           // stop for 2 seconds
  Motion.forw_left();    // turn left
  delay(3000);
  Motion.stop();
 }

void loop()
{

}
```

B. HEADER FILE

```
#ifndef Motors_h
#define Motors_h
#include "Arduino.h"
class Motors
{
public:
  Motors() {
     ENA = 8;   //PWM control pin for right motors
     ENB = 9;  //PWM control pin for left motors
     In_1 = 10;  //1st direction control pin for right motors
       In_2 = 11;  //2nd direction control pin for right motors
     In_3 = 12;  //1st direction control pin for left motors
       In_4 = 13;  //2nd direction control pin for left motors

  }
```

```
  void initialize();
  void forw();
  void back();
  void forward();
  void backward();
  void stop();
  void speed_up();
  void slow_down();
  void forw_right();
  void forw_left();

private:
             int ENA;
       int ENB;
       int In_1;
       int In_2;
       int In_3;
       int In_4;
};
#endif
```

C. CPP FILE

```
1.    #include"Arduino.h"
2.    #include"Motors.h"
3.    #include "SPI.h"
4.
5.    void Motors::initialize()
6.    {
7.      delay(5000);
8.      //Set control pins to be outputs
```

```
9.      pinMode(In_1, OUTPUT);
10.     pinMode(In_2, OUTPUT);
11.     pinMode(In_3, OUTPUT);
12.     pinMode(In_4, OUTPUT);
13.     pinMode(ENA, OUTPUT);
14.     pinMode(ENB, OUTPUT);
15.   }
16.   void Motors::forw() // motors spinning clockwise
17.   {
18.     digitalWrite(In_1, LOW);
19.     digitalWrite(In_2, HIGH);
20.     digitalWrite(In_3, LOW);
21.     digitalWrite(In_4, HIGH);
22.   }
23.   void Motors::back() // motors spinning counter-clockwise
24.   {
25.     digitalWrite(In_1, HIGH);
26.     digitalWrite(In_2, LOW);
27.     digitalWrite(In_3, HIGH);
28.     digitalWrite(In_4, LOW);
29.   }
30.   void Motors::forward() //full speed forward
31.   {
32.     digitalWrite(In_1, LOW);
33.     digitalWrite(In_2, HIGH);
```

```
34.     digitalWrite(In_3, LOW);
35.     digitalWrite(In_4, HIGH);
36.     analogWrite(ENA, 255);     //set right motors to run at
100% duty cycle
37.     analogWrite(ENB, 255);     //set left motors to run at
100% duty cycle
38.   }
39.   void Motors::backward() //full speed backward
40.   {
41.     digitalWrite(In_1, HIGH);
42.     digitalWrite(In_2, LOW);
43.     digitalWrite(In_3, HIGH);
44.     digitalWrite(In_4, LOW);
45.     analogWrite(ENA, 255);    //set right motors to run at
100% duty cycle
46.     analogWrite(ENB, 255);   //set left motors to run at 100%
duty cycle
47.   }
48.   void Motors::stop() //stop
49.   {
50.     analogWrite(ENA, 0);     //set right motors to run at 0%
duty cycle
51.     analogWrite(ENB, 0);     //set left motors to run at 0%
duty cycle
52.   }
53.
```

```
54.   void Motors::speed_up()
55.   {
56.     // fade in from min to max in increments of 10 points:
57.     for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=10)
58.     {
59.        // sets the value (range from 0 to 255):
60.       analogWrite(ENA, fadeValue);
61.       analogWrite(ENB, fadeValue);
62.       // wait for 39.2 milliseconds to see the dimming effect
63.       delay(39.2);
64.     }
65.   }
66.
67.   void Motors::slow_down()
68.   {
69.     // fade out from max to min in increments of 10 points:
70.     for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=10)
71.     {
72.       // sets the value (range from 0 to 255):
73.       analogWrite(ENA, fadeValue);
74.       analogWrite(ENB, fadeValue);
75.       // wait for 39.2 milliseconds to see the dimming effect
76.       delay(39.2);
77.     }
78.   }
```

```
79.   void Motors::forw_right() //forward turn to the right
80.   {
81.     digitalWrite(In_1, HIGH);
82.     digitalWrite(In_2, LOW);
83.     digitalWrite(In_3, LOW);
84.     digitalWrite(In_4, HIGH);
85.     analogWrite(ENA, 200);
86.     analogWrite(ENB, 200);
87.   }
88.
89.   void Motors::forw_left() //forward turn to the right
90.   {
91.     digitalWrite(In_1, LOW);
92.     digitalWrite(In_2, HIGH);
93.     digitalWrite(In_3, HIGH);
94.     digitalWrite(In_4, LOW);
95.     analogWrite(ENA, 200);
96.     analogWrite(ENB, 200);
97.   }
```

D. KEYWORDS FILE

```
Motors      KEYWORD1
initialize  KEYWORD2
forw  KEYWORD2
back  KEYWORD2
forward     KEYWORD2
backward    KEYWORD2
```

```
stop  KEYWORD2
speed_up    KEYWORD2
slow_down   KEYWORD2
forw_right  KEYWORD2
forw_left   KEYWORD2
```

APPENDIX C. TRAJECTORY-FOLLOWER-ROBOT SCRIPTS

The Arduino scripts in this appendix depict the algorithm adopted to perform the trajectory-following task. These scripts contain all the coding necessary to keep track of the robot position while moving from one uploaded waypoint to another.

A. MAIN CODE

```
#include <Destination.h>
#include <SPI.h>
Destination trajectory;
typedef unsigned long Time;
void setup() {
  trajectory.init_1();
  Serial.begin(115200);
  Serial.println();
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE3);
  SPI.setClockDivider(SPI_CLOCK_DIV16);
  trajectory.reset();
  trajectory.init_2();
  trajectory.initialize();
 trajectory.decision();
}
void loop() {
 }
```

B. HEADER FILE

```
#ifndef Destination_h
#define Destination_h
#include "Arduino.h"
// Register Map for the ADNS3080 Optical Optical Flow Sensor
```

```
#define ADNS3080_PRODUCT_ID            0x00
#define ADNS3080_REVISION_ID           0x01
#define ADNS3080_MOTION                0x02
#define ADNS3080_DELTA_X               0x03
#define ADNS3080_DELTA_Y               0x04
#define ADNS3080_SQUAL                 0x05
#define ADNS3080_PIXEL_SUM             0x06
#define ADNS3080_MAXIMUM_PIXEL         0x07
#define ADNS3080_CONFIGURATION_BITS    0x0a
#define ADNS3080_EXTENDED_CONFIG       0x0b
#define ADNS3080_DATA_OUT_LOWER        0x0c
#define ADNS3080_DATA_OUT_UPPER        0x0d
#define ADNS3080_SHUTTER_LOWER         0x0e
#define ADNS3080_SHUTTER_UPPER         0x0f
#define ADNS3080_FRAME_PERIOD_LOWER    0x10
#define ADNS3080_FRAME_PERIOD_UPPER    0x11
#define ADNS3080_MOTION_CLEAR          0x12
#define ADNS3080_FRAME_CAPTURE         0x13
#define ADNS3080_SROM_ENABLE           0x14
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_LOWER      0x19
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_UPPER      0x1a
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_LOWER      0x1b
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_UPPER      0x1c
#define ADNS3080_SHUTTER_MAX_BOUND_LOWER           0x1e
#define ADNS3080_SHUTTER_MAX_BOUND_UPPER           0x1e
#define ADNS3080_SROM_ID               0x1f
#define ADNS3080_OBSERVATION           0x3d
#define ADNS3080_INVERSE_PRODUCT_ID    0x3f
#define ADNS3080_PIXEL_BURST           0x40
#define ADNS3080_MOTION_BURST          0x50
#define ADNS3080_SROM_LOAD             0x60
#define FRAME_LENGTH 900
```

```
class Destination
{
public:
  Destination() {

        ADNS3080_CHIP_SELECT = 4; // chip select pin
        ADNS3080_RESET = 5; // chip reset pin
        ADNS3080_POWER_DOWN = 6; // Power down pin
        ENA = 8;   //PWM control pin for right motors
      ENB = 9;  //PWM control pin for left motors
      In_1 = 10;  //1st direction control pin for right motors
        In_2 = 11;  //2nd direction control pin for right motors
      In_3 = 12;  //1st direction control pin for left motors
        In_4 = 13;  //2nd direction control pin for left motors
      x_sensor = 0;
        y_sensor = 0;
  }

  void measurement(); // sends values of dx and dy over the serial link
and increments distx and disty respectively by dx and dy
  void reset(); // reset sensor
  void init_1();
  void init_2();
  void initialize();
  void forward();
  void backward();
  void stop();
  void spin_right();
  void spin_left();
  void decision();

private:
```

```
     int ADNS3080_CHIP_SELECT;
       int ADNS3080_RESET;
       int ADNS3080_POWER_DOWN;
     float x_sensor;
     float y_sensor;
       int ENA;
       int ENB;
       int In_1;
       int In_2;
       int In_3;
       int In_4;
};
#endif
```

C. CPP FILE

```
#include"Arduino.h"
#include "SPI.h"
#include"Destination.h"

void Destination::init_1()
{
    pinMode(ADNS3080_CHIP_SELECT,OUTPUT);
      pinMode(ADNS3080_POWER_DOWN,OUTPUT);
      pinMode(ADNS3080_RESET,OUTPUT);
      digitalWrite(ADNS3080_CHIP_SELECT,LOW);
      digitalWrite(ADNS3080_RESET, LOW);
      digitalWrite(ADNS3080_POWER_DOWN,HIGH);
}

void Destination::init_2()
{
  int retry = 0;
```

```
 byte productId = 0;
 byte revisionId = 0;

// PRODUCT ID VERIFICATION
 while( retry < 10 ) {
   delayMicroseconds(75);
   // take the chip select low to select the device
   digitalWrite(ADNS3080_CHIP_SELECT,LOW);
   // send the device the register you want to read:
   SPI.transfer(ADNS3080_PRODUCT_ID);
     // small delay
     delayMicroseconds(75);
     // send a value of 0 to read the first byte returned:
   productId = SPI.transfer(0x00);

           if( productId == 0x17 ) {
             Serial.println("\n Found productId ");
               Serial.println(productId, HEX);;
           }

           else{
           Serial.println("\n False productId ");
           Serial.println(productId, HEX);;
           }
        retry++;
         }
  if(productId != 0x17) {
  delay(100);
  exit(1);
   }

// REVISION ID VERIFICATION
   delayMicroseconds(75);
```

```
   // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT,LOW);
    // send the device the register you want to read:
    SPI.transfer(ADNS3080_REVISION_ID);
      // small delay
      delayMicroseconds(75);
      // send a value of 0 to read the first byte returned:
    revisionId = SPI.transfer(0x00);
    Serial.println("\n rev");
    Serial.print(revisionId, HEX);

// Set resolution to 1600 counts/inch
    delayMicroseconds(75);
      // set the chip select to low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT, LOW);

      // send register address
    SPI.transfer(ADNS3080_CONFIGURATION_BITS | 0x80 );

      // send data
      SPI.transfer(0x10);
}

 void Destination::measurement()
{
    byte motion = 0;
      uint8_t dx = 0;
      uint8_t dy = 0;
      int8_t delta_x = 0;
      int8_t delta_y = 0;
      float DELTA_x = 0;
      float DELTA_y = 0;
      byte SQUAL = 0;
```

```
  delayMicroseconds(75);
  // take the chip select low to select the device
digitalWrite(ADNS3080_CHIP_SELECT,LOW);
// send the device the register you want to read:
SPI.transfer(ADNS3080_MOTION);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
motion = SPI.transfer(0x00);

delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_DELTA_X);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
dx = SPI.transfer(0x00);

  delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_DELTA_Y);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
dy = SPI.transfer(0x00);

  delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_SQUAL);
  // small delay
  delayMicroseconds(75);
```

```
  // send a value of 0 to read the first byte returned:
 SQUAL = SPI.transfer(0x00);

 delta_x= (int8_t)dx;

   delta_y= (int8_t)dy;

 DELTA_x=(float)delta_x/1600.0;
 DELTA_y=(float)delta_y/1600.0;

 DELTA_x=DELTA_x/1.605066;
 DELTA_y=DELTA_y/1.605066;

 DELTA_x=100*DELTA_x;
 DELTA_y=100*DELTA_y;

 x_sensor=x_sensor+DELTA_x; // x position in cm
 y_sensor=y_sensor+DELTA_y; // y position in cm

/*  //Display DELTA
 Serial.print(DELTA_x, DEC);
 Serial.print("                              ");
 Serial.print(DELTA_y, DEC);
 //Display x and y
 Serial.print("                              ");
 Serial.print(x_sensor, DEC);
 Serial.print("                              ");
 Serial.println(y_sensor, DEC);   */
 //Display Quality
 /* Serial.println("\n Quality");
 Serial.print(SQUAL, DEC); */

}
```

```
void Destination::decision()
{
    measurement();
      const float Raduis = 14.7; // cm
      /* float X_path[]={0.00, -100.00, -100.00};
      float Y_path[]={100.00, 0.00, 200.00}; */
      /* float X_path[]={-100.00, -100.00, 0.00};
      float Y_path[]={0.00, 200.00, 100.00}; */
      float X_path[]={-100.00, -100.00, -50.00, 0.00, 0.00 };
      float Y_path[]={100.00, 200.00, 200.00, 300.00, 100.00 };
      /* float X_path[]={0};
      float Y_path[]={200.00}; */
      float x_0 = 0;
      float y_0 = 0;
      float Theta = 0;
    float Theta_deg = 0;

      for(int i = 0; i<5; i++) {
    Serial.print("\n The destination ");
    Serial.print(" is: ");
    Serial.print(X_path[i]);
      Serial.print(" , ");
      Serial.println(Y_path[i]);

   if (((Y_path[i])-y_0) > 0)
   {
      if (((X_path[i])-x_0) > 0)
      {
       Theta = atan2(abs((X_path[i])-x_0),abs((Y_path[i])-y_0));
       Theta_deg= Theta*180/3.14;
       Serial.print("\n Spinning Right ");
     Serial.print(" ");
```

```
      Serial.print(Theta_deg);
        Serial.print(" ");
        Serial.println("Degrees");
      while ( (abs(x_sensor) < (Theta*Raduis) ) )
          {
              measurement();
        spin_right();
          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Moving toward destination ");
        while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
        {
          measurement();
              forward();
          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.print("\n Spinning Left ");
      Serial.print(" ");
      Serial.print(Theta_deg);
        Serial.print(" ");
        Serial.println("Degrees");
        while ( (abs(x_sensor) < (Theta*Raduis) ) )
          {
              measurement();
        spin_left();
          }
```

```
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Destination reached ");
       }

       else if (((X_path[i])-x_0) < 0)
       {
        Theta = atan2(abs((X_path[i])-x_0),abs((Y_path[i])-y_0));
        Theta_deg= Theta*180/3.14;
        Serial.print("\n Spinning Left ");
      Serial.print(" ");
      Serial.print(Theta_deg);
        Serial.print(" ");
        Serial.println("Degrees");
      while ( (abs(x_sensor) < (Theta*Raduis) ) )
          {
              measurement();
          spin_left();
          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Moving toward destination ");
        while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
        {
          measurement();
              forward();
          }
        stop();
```

```
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.print("\n Spinning Right ");
      Serial.print(" ");
      Serial.print(Theta_deg);
        Serial.print(" ");
        Serial.println("Degrees");
        while ( (abs(x_sensor) < (Theta*Raduis) ) )
          {
              measurement();
        spin_right();
          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Destination reached ");
       }

       else
       {
        Serial.println(" Moving toward destination ");
             while      (   abs(y_sensor)   <   sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
        {
          measurement();
              forward();
          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
```

```
       Serial.println(" Destination reached ");
      }
  }
////////////////////////////////////////////////////////////////////////
////

   else if (((Y_path[i])-y_0) < 0)
   {
      if (((X_path[i])-x_0) > 0)
      {
       Theta = atan2(abs((Y_path[i])-y_0),abs((X_path[i])-x_0));
       Theta_deg= Theta*(180/3.14)+90;
       Serial.print("\n Spinning Right ");
     Serial.print(" ");
     Serial.print(Theta_deg);
       Serial.print(" ");
       Serial.println("Degrees");
     while ( (abs(x_sensor) < ((Theta+(3.14/2))*Raduis) ) )
           {
            measurement();
        spin_right();
           }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println(" Moving toward destination ");
       while       (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
        {
         measurement();
             forward();
          }
```

```
  stop();
  delay(2000);
  x_sensor = 0;
  y_sensor = 0;
  Serial.print("\n Spinning Left ");
Serial.print(" ");
Serial.print(Theta_deg);
  Serial.print(" ");
  Serial.println("Degrees");
  while ( (abs(x_sensor) < ((Theta+(3.14/2))*Raduis) ) )
       {
        measurement();
    spin_left();
       }
  stop();
  delay(2000);
  x_sensor = 0;
  y_sensor = 0;
  Serial.println(" Destination reached ");
 }

 else if (((X_path[i])-x_0) < 0)
 {
Theta = atan2(abs((Y_path[i])-y_0),abs((X_path[i])-x_0));
  Theta_deg= Theta*(180/3.14)+90;
  Serial.print("\n Spinning Left ");
Serial.print(" ");
Serial.print(Theta_deg);
  Serial.print(" ");
  Serial.println("Degrees");
  while ( (abs(x_sensor) < ((Theta+(3.14/2))*Raduis) ) )
       {
        measurement();
```

```
          spin_left();
             }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Moving toward destination ");
        while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
         {
          measurement();
              forward();
           }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.print("\n Spinning Right ");
      Serial.print(" ");
      Serial.print(Theta_deg);
        Serial.print(" ");
        Serial.println("Degrees");
        while ( (abs(x_sensor) < ((Theta+(3.14/2))*Raduis) ) )
             {
              measurement();
        spin_right();
             }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Destination reached ");
       }
```

```
      else
      {
       Serial.println("\n Spinning Left 180 Degrees ");
       while ( (abs(x_sensor) < (3.14*Raduis) ) )
            {
             measurement();
         spin_left();
            }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println(" Moving toward destination ");
       while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
       {
         measurement();
             forward();
         }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println("\n Spinning Right 180 Degrees ");
       while ( (abs(x_sensor) < (3.14*Raduis) ) )
            {
             measurement();
         spin_right();
            }
       stop();
       delay(2000);
       x_sensor = 0;
```

```
       y_sensor = 0;
       Serial.println(" Destination reached ");
      }
  }
////////////////////////////////////////////////////////////////////////
////

  else
   {
      if (((X_path[i])-x_0) > 0)
      {
     Serial.println("\n Spinning Right 90 Degrees ");
       while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
            {
             measurement();
         spin_right();    // spin right 90 degrees
            }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println(" Moving toward destination ");
       while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
       {
         measurement();
             forward();
         }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println("\n Spinning Left 90 Degrees ");
```

```
       while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
            {
             measurement();
         spin_left();    // spin left 90 degrees
            }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println(" Destination reached ");
      }

      else if (((X_path[i])-x_0) < 0)
      {
     Serial.print("\n Spinning Left 90 Degrees ");
       while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
            {
             measurement();
         spin_left();    // spin left 90 degrees
            }
       stop();
       delay(2000);
       x_sensor = 0;
       y_sensor = 0;
       Serial.println(" Moving toward destination ");
       while        (    abs(y_sensor)    <    sqrt(pow(abs((X_path[i])-
x_0),2)+pow(abs((Y_path[i])-y_0),2)) )
       {
         measurement();
             forward();
         }
       stop();
       delay(2000);
```

```
        x_sensor = 0;
        y_sensor = 0;
        Serial.print("\n Spinning Right 90 Degrees ");
        while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
             {
              measurement();
          spin_right();    // spin right 90 degrees
             }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Destination reached ");
       }

       else
       {
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Same Last Destination ");
        Serial.println(" Destination reached ");
       }
  }
  x_0 = X_path[i];
  y_0 = Y_path[i];

  }

}

void Destination::reset()
```

```
{
    digitalWrite(ADNS3080_RESET,HIGH);                 // reset sensor
      delayMicroseconds(10);
      digitalWrite(ADNS3080_RESET,LOW);                       // return
sensor to normal
}

void Destination::initialize()
{
  delay(5000);
  //Set control pins to be outputs
  pinMode(In_1, OUTPUT);
  pinMode(In_2, OUTPUT);
  pinMode(In_3, OUTPUT);
  pinMode(In_4, OUTPUT);
  pinMode(ENA, OUTPUT);
  pinMode(ENB, OUTPUT);

}

void Destination::forward()
{
  digitalWrite(In_1, LOW);
  digitalWrite(In_2, HIGH);
  digitalWrite(In_3, LOW);
  digitalWrite(In_4, HIGH);
  analogWrite(ENA, 117);    //set right motors to run at duty cycle
  analogWrite(ENB, 104);    //set left motors to run at  duty cycle
  /* if  ( x_pos > 0 )
      {
        analogWrite(ENA, 145);
        analogWrite(ENB, 150);
      }
```

```
     else if  ( x_pos < 0 )
      {
          analogWrite(ENA, 150);
        analogWrite(ENB, 130);
        }
       else
       {
        analogWrite(ENA, 150);
        analogWrite(ENB, 130);
       } */

}

void Destination::backward()
{
  digitalWrite(In_1, HIGH);
  digitalWrite(In_2, LOW);
  digitalWrite(In_3, HIGH);
  digitalWrite(In_4, LOW);
  analogWrite(ENA, 115);
  analogWrite(ENB, 106);
}

void Destination::stop() //stop
{
  analogWrite(ENA, 0);    //set right motors to run at 0% duty cycle
  analogWrite(ENB, 0);    //set left motors to run at 0% duty cycle
}

void Destination::spin_right() //forward turn to the right
{
  digitalWrite(In_1, HIGH);
  digitalWrite(In_2, LOW);
```

```
  digitalWrite(In_3, LOW);
  digitalWrite(In_4, HIGH);
  analogWrite(ENA, 200);
  analogWrite(ENB, 200);
}

void Destination::spin_left() //forward turn to the right
{
  digitalWrite(In_1, LOW);
  digitalWrite(In_2, HIGH);
  digitalWrite(In_3, HIGH);
  digitalWrite(In_4, LOW);
  analogWrite(ENA, 200);
  analogWrite(ENB, 200);
 }
```

D. KEYWORDS FILE

```
Destination KEYWORD1
initialize  KEYWORD2
forward     KEYWORD2
backward    KEYWORD2
stop  KEYWORD2
spin_right  KEYWORD2
spin_left   KEYWORD2
measurement KEYWORD2
reset KEYWORD2
init_1      KEYWORD2
init_2      KEYWORD2
decision    KEYWORD2
```

APPENDIX D. OBSTACLE DETECTION AND AVOIDANCE ROBOT SCRIPTS

The Arduino scripts in this appendix depict the algorithm adopted to perform the obstacle detection and avoidance tasks. These scripts contain all the coding necessary to move the robot from one point to another avoiding collision with obstacles on the way.

A. MAIN CODE

```
#include <Destination.h>
#include <SPI.h>
#include <Servo.h>
Destination trajectory;
typedef unsigned long Time;

void setup() {
  trajectory.init_1();
  Serial.begin(115200);
  Serial.println();
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE3);
  SPI.setClockDivider(SPI_CLOCK_DIV16);
  trajectory.reset();
  trajectory.init_2();
  trajectory.initialize();
  trajectory.decision();
}

void loop() {
 }
```

B. HEADER FILE

```
#ifndef Destination_h
#define Destination_h
#include "Arduino.h"
// Register Map for the ADNS3080 Optical Optical Flow Sensor
#define ADNS3080_PRODUCT_ID            0x00
#define ADNS3080_REVISION_ID           0x01
#define ADNS3080_MOTION                0x02
#define ADNS3080_DELTA_X               0x03
#define ADNS3080_DELTA_Y               0x04
#define ADNS3080_SQUAL                 0x05
#define ADNS3080_PIXEL_SUM             0x06
#define ADNS3080_MAXIMUM_PIXEL         0x07
#define ADNS3080_CONFIGURATION_BITS    0x0a
#define ADNS3080_EXTENDED_CONFIG       0x0b
#define ADNS3080_DATA_OUT_LOWER        0x0c
#define ADNS3080_DATA_OUT_UPPER        0x0d
#define ADNS3080_SHUTTER_LOWER         0x0e
#define ADNS3080_SHUTTER_UPPER         0x0f
#define ADNS3080_FRAME_PERIOD_LOWER    0x10
#define ADNS3080_FRAME_PERIOD_UPPER    0x11
#define ADNS3080_MOTION_CLEAR          0x12
#define ADNS3080_FRAME_CAPTURE         0x13
#define ADNS3080_SROM_ENABLE           0x14
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_LOWER      0x19
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_UPPER      0x1a
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_LOWER      0x1b
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_UPPER      0x1c
#define ADNS3080_SHUTTER_MAX_BOUND_LOWER           0x1e
#define ADNS3080_SHUTTER_MAX_BOUND_UPPER           0x1e
#define ADNS3080_SROM_ID               0x1f
#define ADNS3080_OBSERVATION           0x3d
```

```
#define ADNS3080_INVERSE_PRODUCT_ID    0x3f
#define ADNS3080_PIXEL_BURST           0x40
#define ADNS3080_MOTION_BURST          0x50
#define ADNS3080_SROM_LOAD             0x60
#define FRAME_LENGTH 900

class Destination
{
public:
  Destination() {

       ADNS3080_CHIP_SELECT_1 = 4; // chip select pin
       ADNS3080_CHIP_SELECT_2 = 2; // chip select pin
       ADNS3080_RESET = 5; // chip reset pin
       ADNS3080_POWER_DOWN = 6; // Power down pin
       Servo_pwm = 3; //PWM control pin for servo
       ENA = 8;   //PWM control pin for right motors
      ENB = 9;  //PWM control pin for left motors
     In_1 = 10;  //1st direction control pin for right motors
     In_2 = 11;  //2nd direction control pin for right motors
     In_3 = 12;  //1st direction control pin for left motors
     In_4 = 13;  //2nd direction control pin for left motors
     x_sensor = 0;
       y_sensor = 0;
       QUAL2 = 0;
  }

  void measurement(); // sends values of dx and dy over the serial link
and increments distx and disty respectively by dx and dy (sensor 1)
  void measurement2(); // sends values of dx and dy over the serial
link and increments distx and disty respectively by dx and dy (sensor
2)
  void reset(); // reset sensor
```

```
  void init_1();
  void init_2();
  void initialize();
  void forward();
  void backward();
  void stop();
  void spin_right();
  void spin_left();
  void decision();

private:

     int ADNS3080_CHIP_SELECT_1;
       int ADNS3080_CHIP_SELECT_2;
       int ADNS3080_RESET;
       int ADNS3080_POWER_DOWN;
     float x_sensor;
     float y_sensor;
       int QUAL2;
       int ENA;
       int Servo_pwm;
       int ENB;
       int In_1;
       int In_2;
       int In_3;
       int In_4;

};

#endif
```

C. CPP FILE

```
#include "Arduino.h"
```

```
#include "SPI.h"
#include "Destination.h"
#include "Servo.h"

void Destination::init_1()
{
    pinMode(ADNS3080_CHIP_SELECT_1,OUTPUT);
      pinMode(ADNS3080_CHIP_SELECT_2,OUTPUT);
      pinMode(ADNS3080_POWER_DOWN,OUTPUT);
      pinMode(ADNS3080_RESET,OUTPUT);

      digitalWrite(ADNS3080_CHIP_SELECT_1,LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_2,HIGH);
      digitalWrite(ADNS3080_RESET, LOW);
      digitalWrite(ADNS3080_POWER_DOWN,HIGH);

      digitalWrite(ADNS3080_CHIP_SELECT_2,LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_1,HIGH);
      digitalWrite(ADNS3080_RESET, LOW);
      digitalWrite(ADNS3080_POWER_DOWN,HIGH);
}

void Destination::init_2()
{
  // PRODUCT ID VERIFICATION for sensor 1
  int retry_1 = 0;
  byte productId_1 = 0;

   while( retry_1 < 10 ) {
    delayMicroseconds(75);
    // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT_1,LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_2,HIGH);
```

```
  // send the device the register you want to read:
  SPI.transfer(ADNS3080_PRODUCT_ID);
    // small delay
    delayMicroseconds(75);
    // send a value of 0 to read the first byte returned:
  productId_1 = SPI.transfer(0x00);

          if( productId_1 == 0x17 ) {
            Serial.println("\n Found productId for sensor 1 ");
              Serial.println(productId_1, HEX);
          }

          else{
          Serial.println("\n False productId for sensor 1 ");
          Serial.println(productId_1, HEX);
          }
       retry_1++;
        }
 if(productId_1 != 0x17) {
 delay(100);
 exit(1);
  }

 // PRODUCT ID VERIFICATION for sensor 2
int retry_2 = 0;
byte productId_2 = 0;

    while( retry_2 < 10 ) {
  delayMicroseconds(75);
  // take the chip select low to select the device
  digitalWrite(ADNS3080_CHIP_SELECT_2,LOW);
    digitalWrite(ADNS3080_CHIP_SELECT_1,HIGH);
  // send the device the register you want to read:
```

```
    SPI.transfer(ADNS3080_PRODUCT_ID);
      // small delay
      delayMicroseconds(75);
      // send a value of 0 to read the first byte returned:
    productId_2 = SPI.transfer(0x00);

            if( productId_2 == 0x17 ) {
              Serial.println("\n Found productId for sensor 2 ");
                Serial.println(productId_2, HEX);
            }

            else{
            Serial.println("\n False productId for sensor 2 ");
            Serial.println(productId_2, HEX);
            }
          retry_2++;
           }
    if(productId_2 != 0x17) {
   delay(100);
   exit(1);
    }

// Set sensor 1 resolution to 1600 counts/inch
    delayMicroseconds(75);
      // set the chip select to low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT_1, LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_2,HIGH);

      // send register address
    SPI.transfer(ADNS3080_CONFIGURATION_BITS | 0x80 );

      // send data
      SPI.transfer(0x10);
```

```
// Set sensor 2 resolution to 1600 counts/inch
    delayMicroseconds(75);
      // set the chip select to low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT_2, LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_1,HIGH);

      // send register address
    SPI.transfer(ADNS3080_CONFIGURATION_BITS | 0x80 );

      // send data
      SPI.transfer(0x10);
}

 void Destination::measurement()
{
    byte motion = 0;
      uint8_t dx = 0;
      uint8_t dy = 0;
      int8_t delta_x = 0;
      int8_t delta_y = 0;
      float DELTA_x = 0;
      float DELTA_y = 0;
      // byte SQUAL = 0;
      uint8_t SQUAL1 = 0;

      delayMicroseconds(75);
      // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT_1,LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_2,HIGH);
    // send the device the register you want to read:
    SPI.transfer(ADNS3080_MOTION);
      // small delay
```

```
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
motion = SPI.transfer(0x00);

delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_DELTA_X);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
dx = SPI.transfer(0x00);

  delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_DELTA_Y);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
dy = SPI.transfer(0x00);

  delayMicroseconds(75);
  // send the device the register you want to read:
SPI.transfer(ADNS3080_SQUAL);
  // small delay
  delayMicroseconds(75);
  // send a value of 0 to read the first byte returned:
SQUAL1 = SPI.transfer(0x00);

delta_x= (int8_t)dx;

  delta_y= (int8_t)dy;

DELTA_x=(float)delta_x/1600.0;
```

```
    DELTA_y=(float)delta_y/1600.0;

    DELTA_x=DELTA_x/1.605066;
    DELTA_y=DELTA_y/1.605066;

    DELTA_x=100*DELTA_x;
    DELTA_y=100*DELTA_y;

    x_sensor=x_sensor+DELTA_x; // x position in cm
    y_sensor=y_sensor+DELTA_y; // y position in cm

    //Display DELTA
    Serial.print(DELTA_x, DEC);
    Serial.print("                              ");
    Serial.print(DELTA_y, DEC);
    //Display x and y
    Serial.print("                              ");
    Serial.print(x_sensor, DEC);
    Serial.print("                              ");
    Serial.println(y_sensor, DEC);
    //Display Quality
    Serial.println("Quality");
    Serial.println(SQUAL1, DEC);

}

void Destination::measurement2()
{

      // byte SQUAL = 0;
      uint8_t SQUAL2 = 0;

      delayMicroseconds(75);
```

```
      // take the chip select low to select the device
    digitalWrite(ADNS3080_CHIP_SELECT_2,LOW);
      digitalWrite(ADNS3080_CHIP_SELECT_1,HIGH);

      // send the device the register you want to read:
    SPI.transfer(ADNS3080_SQUAL);
      // small delay
      delayMicroseconds(75);
      // send a value of 0 to read the first byte returned:
    SQUAL2 = SPI.transfer(0x00);

    //Display Quality
    Serial.println("\n Quality");
    Serial.print(SQUAL2, DEC);
      QUAL2= SQUAL2;

}




void Destination::decision()
{
    measurement();
      measurement2();
      const float Raduis = 14.7; // cm
      float X_path[]={0.00 };
      float Y_path[]={300.00};
      float x1 = 0;
      float y1 = 0;
      float y2 = 0;
      float Theta = 0;
    float Theta_deg = 0;
```

```
  Servo myservo;  // create servo object to control a servo
myservo.attach(Servo_pwm);  // attaches the servo on pin 9 to the
servo object
  myservo.write(85);

  for(int i = 0; i<1; i++) {
Serial.print("\n The destination ");
Serial.print(" is: ");
Serial.print(X_path[i]);
  Serial.print(" , ");
  Serial.println(Y_path[i]);


  Serial.println(" Moving toward destination ");
  while  ( Y_path[i] > (y1+y2) )
   {
     measurement();
         measurement2();
         if (QUAL2!=0)
         {
           measurement();
           measurement2();
           forward();
           y1=abs(y_sensor);
         }
         else
         {
              stop();
              delay(2000);
              myservo.write(0);
        delay(3000);
              measurement2();
              if (QUAL2!=0)
```

```
 {
        x_sensor = 0;
  y_sensor = 0;
        myservo.write(85);
        delay(2000);
        Serial.print("\n Spinning Right ");
Serial.print(" ");
Serial.print(90);
  Serial.print(" ");
  Serial.println("Degrees");
        while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
       {
          measurement();
      spin_right();
       }
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        myservo.write(175);
        delay(3000);
        Serial.print("\n Moving forward ");
        measurement2();
        while(QUAL2<50)
     {
          measurement();
                measurement2();
      forward();
                x1=abs(y_sensor);
                }
        stop();
  delay(2000);
        x_sensor = 0;
```

```
  y_sensor = 0;
         while(abs(y_sensor) < 30)
     {
          measurement();
                measurement2();
      forward();
                }
x1=x1+30;
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        myservo.write(85);
        Serial.print("\n Spinning Left ");
Serial.print(" ");
Serial.print(90);
  Serial.print(" ");
  Serial.println("Degrees");
        while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
       {
          measurement();
      spin_left();
       }
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        while ( abs(y_sensor) < (Y_path[i]-y1) )
        {
              measurement();
        forward();
        }
        stop();
```

```
  y2=abs(y_sensor);
        delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        Serial.print("\n Spinning Left ");
Serial.print(" ");
Serial.print(90);
  Serial.print(" ");
  Serial.println("Degrees");
        while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
       {
          measurement();
      spin_left();
       }
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        Serial.print("\n Moving forward ");
        measurement();
        while(abs(y_sensor)< x1)
     {
          measurement();
                forward();
                }
        stop();
        delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        Serial.print("\n Spinning Right ");
Serial.print(" ");
Serial.print(90);
  Serial.print(" ");
```

```
 Serial.println("Degrees");
       while ( (abs(x_sensor) < ((3.14/2)*Raduis) ) )
      {
         measurement();
     spin_right();
      }

}

else
{
        stop();
          myservo.write(85);
          delay(2000);
        myservo.write(175);
  delay(2000);
        measurement2();
        if (QUAL2!=0)
          {
           stop();
             x_sensor = 0;
     y_sensor = 0;
             Serial.print("\n Spinning left ");
   Serial.print(" ");
   Serial.print(90);
     Serial.print(" ");
     Serial.println("Degrees");
          while ( (abs(x_sensor) < ((3.14/2)*Raduis)
) )
      {
         measurement();
     spin_left();
      }
```

```
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        myservo.write(0);
        delay(3000);
        Serial.print("\n Moving forward ");
        measurement2();
        while(QUAL2<50)
 {
      measurement();
            measurement2();
  forward();
            x1=abs(y_sensor);
            }
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        while(abs(y_sensor) < 30)
 {
       measurement();
             measurement2();
   forward();
            }
x1=x1+30;
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        myservo.write(85);
        Serial.print("\n Spinning Right ");
Serial.print(" ");
```

```
                        Serial.print(90);
                          Serial.print(" ");
                          Serial.println("Degrees");
                                while ( (abs(x_sensor) < ((3.14/2)*Raduis)
) )
                           {
                              measurement();
                          spin_right();
                           }
                                stop();
                          delay(2000);
                                x_sensor = 0;
                          y_sensor = 0;
                                while ( abs(y_sensor) < (Y_path[i]-y1) )
                                 {
                                    measurement();
                              forward();
                                 }
                                stop();
                          y2=abs(y_sensor);
                                delay(2000);
                                x_sensor = 0;
                          y_sensor = 0;
                                Serial.print("\n Spinning Right ");
                        Serial.print(" ");
                        Serial.print(90);
                          Serial.print(" ");
                          Serial.println("Degrees");
                                while ( (abs(x_sensor) < ((3.14/2)*Raduis)
) )
                           {
                               measurement();
                           spin_right();
```

```
   }
        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        Serial.print("\n Moving forward ");
        measurement();
        while(abs(y_sensor)< x1)
 {
     measurement();
          forward();
          }

        stop();
  delay(2000);
        x_sensor = 0;
  y_sensor = 0;
        Serial.print("\n Spinning left ");
Serial.print(" ");
Serial.print(90);
  Serial.print(" ");
  Serial.println("Degrees");
        while ( (abs(x_sensor) < ((3.14/2)*Raduis)
) )
   {
      measurement();
  spin_left();
   }
        stop();
  delay(2000);
  x_sensor = 0;
  y_sensor = 0;
       }
```

```
                        else
                        {
                              stop();
                             x_sensor = 0;
                      y_sensor = 0;
                                myservo.write(85);
                            delay(2000);
                            Serial.print("\n Spinning Right ");
                    Serial.print(" ");
                    Serial.print(90);
                      Serial.print(" ");
                      Serial.println("Degrees");
                            while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
                       {
                          measurement();
                      spin_right();
                       }
                             stop();
                       delay(2000);
                             x_sensor = 0;
                       y_sensor = 0;
                                  delay(2000);
                             Serial.print("\n Spinning Right ");
                     Serial.print(" ");
                     Serial.print(90);
                      Serial.print(" ");
                      Serial.println("Degrees");
                            while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
                      {
```

```
    measurement();
 spin_right();
  }
       stop();
  delay(2000);
       x_sensor = 0;
  y_sensor = 0;
           myservo.write(175);
       delay(3000);
       Serial.print("\n Moving forward ");
       measurement2();
       while(QUAL2<50)
{
    measurement();
          measurement2();
 forward();
         }
       y1=y1-abs(y_sensor);
           stop();
  delay(2000);
       x_sensor = 0;
  y_sensor = 0;
       while(abs(y_sensor) < 30)
{
    measurement();
          measurement2();
 forward();
         }
y1=y1-30;
       stop();
  delay(2000);
       x_sensor = 0;
  y_sensor = 0;
```

```
                myservo.write(85);
                Serial.print("\n Spinning Left ");
            Serial.print(" ");
            Serial.print(90);
              Serial.print(" ");
              Serial.println("Degrees");
                    while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
              {
                  measurement();
              spin_left();
              }
                    stop();
              delay(2000);
                    x_sensor = 0;
              y_sensor = 0;
                       myservo.write(175);
                    delay(3000);
                    Serial.print("\n Moving forward ");
                    measurement2();
                    while(QUAL2<50)
            {
                 measurement();
                       measurement2();
             forward();
                       x1=abs(y_sensor);
                      }
                    stop();
              delay(2000);
                    x_sensor = 0;
              y_sensor = 0;
                    while(abs(y_sensor) < 30)
            {
```

```
                measurement();
                      measurement2();
             forward();
                      }
            x1=x1+30;
                   stop();
              delay(2000);
                   x_sensor = 0;
              y_sensor = 0;
                   myservo.write(85);
                   Serial.print("\n Spinning Left ");
            Serial.print(" ");
            Serial.print(90);
              Serial.print(" ");
              Serial.println("Degrees");
                   while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
              {
                  measurement();
              spin_left();
              }
                        stop();
              delay(2000);
                   x_sensor = 0;
              y_sensor = 0;
                        while     (     abs(y_sensor)     <
(Y_path[i]-y1) )
                   {
                        measurement();
                  forward();
                   }
                    stop();
             y2=abs(y_sensor);
```

```
                delay(2000);
                x_sensor = 0;
              y_sensor = 0;
                Serial.print("\n Spinning Left ");
             Serial.print(" ");
             Serial.print(90);
              Serial.print(" ");
              Serial.println("Degrees");
                   while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
              {
                 measurement();
             spin_left();
              }
                   stop();
              delay(2000);
                   x_sensor = 0;
              y_sensor = 0;
                   Serial.print("\n Moving forward ");
                   measurement();
                   while(abs(y_sensor)< x1)
            {
                 measurement();
                     forward();
                    }
                   stop();
                   delay(2000);
                   x_sensor = 0;
              y_sensor = 0;
                   Serial.print("\n Spinning Right ");
            Serial.print(" ");
            Serial.print(90);
              Serial.print(" ");
```

```
                          Serial.println("Degrees");
                                while      (      (abs(x_sensor)      <
((3.14/2)*Raduis) ) )
                          {
                             measurement();
                          spin_right();
                          }
                             }

                  }

              }

          }
        stop();
        delay(2000);
        x_sensor = 0;
        y_sensor = 0;
        Serial.println(" Destination reached ");

  }

}


void Destination::reset()
{
    digitalWrite(ADNS3080_RESET,HIGH);                 // reset sensor
      delayMicroseconds(10);
      digitalWrite(ADNS3080_RESET,LOW);                        // return
sensor to normal
}
```

```
void Destination::initialize()
{
   delay(5000);
   //Set control pins to be outputs
  pinMode(In_1, OUTPUT);
  pinMode(In_2, OUTPUT);
  pinMode(In_3, OUTPUT);
  pinMode(In_4, OUTPUT);
  pinMode(ENA, OUTPUT);
  pinMode(ENB, OUTPUT);

  /* Servo myservo;  // create servo object to control a servo
                // a maximum of eight servo objects can be created
   myservo.attach(Servo_pwm);   // attaches the servo on pin 9 to the
servo object
   myservo.write(85);
   delay(3000);
   myservo.write(0);
   delay(3000);
   myservo.write(175);
   delay(3000);
   myservo.write(85);
   delay(3000); */
}

void Destination::forward()
{
  digitalWrite(In_1, LOW);
  digitalWrite(In_2, HIGH);
  digitalWrite(In_3, LOW);
  digitalWrite(In_4, HIGH);
  analogWrite(ENA, 115);    //set right motors to run at duty cycle
  analogWrite(ENB, 106);    //set left motors to run at  duty cycle
```

```
  /* if  ( x_pos > 0 )
      {
        analogWrite(ENA, 145);
        analogWrite(ENB, 150);
      }
     else if  ( x_pos < 0 )
      {
          analogWrite(ENA, 150);
        analogWrite(ENB, 130);
        }
       else
       {
        analogWrite(ENA, 150);
        analogWrite(ENB, 130);
       } */
}
void Destination::backward()
{
  digitalWrite(In_1, HIGH);
  digitalWrite(In_2, LOW);
  digitalWrite(In_3, HIGH);
  digitalWrite(In_4, LOW);
  analogWrite(ENA, 115);
  analogWrite(ENB, 106);
}

void Destination::stop() //stop
{
  analogWrite(ENA, 0);    //set right motors to run at 0% duty cycle
  analogWrite(ENB, 0);    //set left motors to run at 0% duty cycle
}

void Destination::spin_right() //forward turn to the right
```

```
{
  digitalWrite(In_1, HIGH);
  digitalWrite(In_2, LOW);
  digitalWrite(In_3, LOW);
  digitalWrite(In_4, HIGH);
  analogWrite(ENA, 200);
  analogWrite(ENB, 200);
}

void Destination::spin_left() //forward turn to the right
{
  digitalWrite(In_1, LOW);
  digitalWrite(In_2, HIGH);
  digitalWrite(In_3, HIGH);
  digitalWrite(In_4, LOW);
  analogWrite(ENA, 200);
  analogWrite(ENB, 200);
 }
```

D. KEYWORDS FILE

```
Destination KEYWORD1
initialize  KEYWORD2
forward     KEYWORD2
backward    KEYWORD2
stop  KEYWORD2
spin_right  KEYWORD2
spin_left   KEYWORD2
measurement KEYWORD2
measurement2      KEYWORD2
reset KEYWORD2
init_1      KEYWORD2
init_2      KEYWORD2
decision    KEYWORD2
```

LIST OF REFERENCES

[1] H. Chao, Y. Cao, and Y. Q. Chen, "Autopilots for small unmanned aerial vehicles: A survey," *Int. J. Control Autom. Syst.*, vol. 8, no. 1, pp. 36–44, 2010.

[2] S. Baker, D. Scharstein, J. Lewis, S. Roth, M. J. Black, and R. Szeliski, "A database and evaluation methodology for optical flow," *Int. J. Comput. Vis.*, vol. 92, no. 1, pp. 1–31, 2011.

[3] H. Chao, Y. Gu, and M. Napolitano, "A survey of optical flow techniques for robotics navigation applications," *J. Intell Robot. Syst.*, vol. 73, pp. 361–372, 2014.

[4] M. J. Black and P. Anandan, "The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields," *Comp. Vis. Image Underst.*, vol. 63, no. 1, pp. 75–104, 1996.

[5] S. Roth and M. J. Black, "On the spatial statistics of optical flow," *Int. J. Comput. Vis.*, vol. 74, no. 1, pp. 3–50, 2007.

[6] J. Barron, D. Fleet, and S. Beauchemin, "Performance of optical flow techniques," *Int. J. Comput. Vis.*, vol. 12, no. 1, pp. 43–77, 1994.

[7] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proc. DARPA Image Understanding Workshop*, 1981, pp. 121–130.

[8] B. Horn and B. Schunck, "Determining optical flow," *Artif. Intell.*, vol. 17, pp. 185–203, 1981.

[9] A. Bonarini, M. Matteucci, and M. Restelli, "Automatic error detection and reduction for an odometric sensor based on two optical mice," in *Proc. IEEE Int. Conf. Robotics Automation (ICRA)*, Apr. 2005, pp. 1675–1680.

[10] S. Singh and K. Waldron, "Design and evaluation of an integrated planar localization method for desktop robotics," in *Proc. IEEE Int. Conf. Robotics Automation (ICRA)*, May 2004, vol. 2, pp. 1109–1114.

[11] DC motor encoder. (n.d.). *The Tech Resources Wiki*. Available: http://www.stab-iitb.org/wiki/DC_Motor_Encoder. Accessed Aug. 6, 2015.

[12] Rotary encoder. (n.d.). *Wikipedia*. Available: https://en.wikipedia.org/wiki/Rotary_encoder. Accessed Aug. 6, 2015.

[13] Incremental encoders. (2012, Aug. 9). Minarik Automation and Control. [Online]. Available: http://training.minarik.com/drupal/content/articles/incremental-encoders

[14] Servo Motor glossary of terms. (n.d.). Oriental Motor. [Online]. Available: http://www.orientalmotor.com/technology/articles/servo-motor-glossary.html. Accessed Aug. 6, 2015.

[15] J. Bradshaw, C. Lollini, and B. Bishop, "On the development of an enhanced optical mouse sensor for odometry and mobile robotics education," in *Proc. 39th Southeastern Symp. Syst. Theory (SSST '07)*, Mar. 2007, pp. 6–10.

[16] S. Thakoor, J. Morookian, J. Chahl, D. Soccol, B. Hine, and S. Zornetzer, "Insect-inspired optical-flow navigation sensors," NASA Jet Propulsion Laboratory, Pasadena, CA, Tech. Rep. NPO-40173, Oct. 2005.

[17] Avago Technologies. (2008, Oct. 20). ADNS-3080 High-Performance Optical Mouse Sensor. [Online]. Available: http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2009/ncr6_wjw27/ncr6_wjw27/docs/adns_3080.pdf.

[18] Hyperphysics. (n.d.). Image formation by lenses and the eye. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/class/phscilab/imagei.html. Accessed Aug. 6, 2015.

[19] S. Bell, "High-Precision Robot Odometry Using an Array of Optical Mice," in IEEE colloquium, Oklahoma Christian University, 2011.

[20] Dual H-Bridge Motor Driver. *GeekOnFire Wiki.* Available: http://www.geekonfire.com/wiki/index.php?title=Dual_H-Bridge_Motor_Driver. Accessed Aug. 5, 2015.

[21] Arduino. (n.d.). Arduino Due. [Online]. Available: https://www.arduino.cc/en/Main/arduinoBoardDue. Accessed Aug. 5, 2015.

[22] Parallax Inc., "Parallax Standard Servo (#900-00005)." Rocklin, CA: Parallax Inc., 2011.

[23] Digi International Inc., "XBee-PRO 900/DigiMeshTM 900 OEM RF Modules." Minnetonka, MN: Digi International Inc., 2008.

[24] S. Brachmann. (2014, Jan. 20). GE seeks patent on flight control system to more accurately predict fuel usage, arrival time. *IPWatchdog*. [Online]. Available: http://www.ipwatchdog.com/2014/01/20/ge-seeks-patent-on-flight-control-system-to-more-accurately-predict-fuel-usage-arrival-time/id=47580/

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California